BIG PROVENANCE STREAM PROCESSING FOR DATA-INTENSIVE COMPUTATIONS

Isuru Suriarachchi

Submitted to the faculty of the University Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the School of Informatics, Computing and Engineering
Indiana University
November 2018

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Beth Plale, Ph.D.

David Leake, Ph.D.

Ryan Newton, Ph.D.

Judy Qiu, Ph.D.

Date of Defense: October 12, 2018

To my wife Pavithri, daughter Miheli, and our parents.

Acknowledgements

This dissertation is a result of an effort over many years. There are so many people who helped me in various ways during this endeavor. Without their generous support and encouragement, this work would not have been possible.

First of all, I am so grateful to my Ph.D. advisor, Prof. Beth Plale for her invaluable support, guidance, and encouragement throughout my Ph.D. Her research experience over many years across multiple areas of Computer Science helped me in many ways to solve hard research problems and to successfully present them as publications. In addition to that, she was so kind to me and my family during our hard times. I am truly honored to have worked with her throughout my Ph.D. studies.

I would like to thank my research committee members Prof. David Leake, Prof. Ryan Newton and Prof. Judy Qiu for their guidance and advice on my qualifying exams, thesis proposal, and final dissertation. I should thank all professors at the School of Informatics, Computing, and Engineering from whom I took a number of courses which helped immensely to improve my knowledge and skills.

It has been great to work with my past and present peers at the Data to Insight Center including Milinda Pathirage, Peng Chen, Inna Kouper, Yuan Luo, Quan Zhou, Kavitha Chandrasekar, Yu Luo, Charitha Dandeniya Arachchi, Samitha Liyanage, Ratharanjan Kunalan and Sachith Withana. I thank all of them for insightful research discussions and

for all the fun we had together which made a great environment to work in. Also, I should thank all past and present administrative staff members at the center including Mary Nell Shiflet, Jenny Olmes-Stevens and Jodi Stern.

My special gratitude goes to Dr. Sanjiva Weerawarana who encouraged and guided me toward graduate studies and helped me in many ways during the application process. He also was a teacher, mentor and my first CEO at WSO2 where I started my career. I take this opportunity to thank all my past colleagues at WSO2 who also supported in many ways.

I owe my special thanks to all my past teachers at the University of Moratuwa, Sri Lanka and my high school, Richmond College, Galle, Sri Lanka. They all have contributed immensely towards my success and I would not have been here without their support.

The Sri Lankan community in Bloomington has been wonderful and their love and care made our time there a great one. I thank all friends including Supun Kamburugamuve, Lahiru Gunathilake, Saliya Ekanayake, Milinda Pathirage, Udayanga Wickramasinghe, Shammi Jayasinghe, Amila Jayasekara, Thilina Gunarathne, Ratharanjan Kunalan, Wasantha Jayawardene and their families for their immense support. Without them, we would not have survived some of our hardest times away from family.

I have no enough words to thank my parents and two brothers for their unconditional love and encouragement throughout my life. My parents took every effort to create a great environment for me to succeed. I grew up looking at my elder brothers and they have

been role models for me. Here I thank all my close relatives and childhood friends too who helped me in many ways over the years.

Finally, there is one special person who made all of this possible. My lovely wife, Pavithri has been always behind me during our good and bad times. She sacrificed so many things for me and my daughter and took so much pain to make us comfortable. I owe so much to her for making the completion of this thesis a possibility and apologize her for all the responsibilities that I have missed over last six years. My daughter, Miheli who is the most precious gift that I have ever received has also been a great encouragement for me to succeed.

Isuru Suriarachchi

BIG PROVENANCE STREAM PROCESSING FOR DATA-INTENSIVE COMPUTATIONS

Industry, academia, and research alike are grappling with the opportunities that Big Data brings in the ability to analyze data from numerous sources for insight, decision making, and predictive forecasts. The analysis workflows for dealing with such volumes of data are said to be *large scale data-intensive computations* (DICs). Data-intensive computation frameworks, also known as Big Data processing frameworks, carry out both online and offline processing.

Big Data analysis workflows frequently consist of multiple steps: data cleaning, joining data from different sources and applying processing algorithms. Critically today the steps of a given workflow may be performed with different processing frameworks simultaneously, complicating the lifecycle of the data products that go through the workflow. This is particularly the case in emerging Big Data management solutions like Data Lakes in which data from multiple sources are stored in a shared storage solution and analyzed for different purposes at different points of time. In such an environment, accessibility and traceability of data products are known to be hard to achieve.

Data provenance, or data lineage, leads to a good solution for this problem as it provides the derivation history of a data product and helps in monitoring, debugging and reproducing computations. Our initial research produced a provenance-based reference

architecture and a prototype implementation to achieve better traceability and management. Experiments show that the size of fine-grained provenance collected from data-intensive computations can be several times larger than the original data itself, creating a Big Data problem referred to in the literature "Big Provenance". Storing and managing Big Provenance for later analysis is not be feasible for some data-intensive applications due to high resource consumption. In addition to that, not all provenance is equally valuable and can be summarized without loss of critical information.

In this thesis, I apply stream processing techniques to analyze streams of provenance captured from data-intensive computations. The specific contributions are several. First, a provenance model which includes formal definitions for provenance stream, forward provenance and backward provenance in the context of data-intensive computations. Second, a stateful, one-pass, parallel stream processing algorithm to summarize a full provenance stream on-the-fly by preserving backward provenance and forward provenance. The algorithm is resilient to provenance events arriving out-of-order. Multiple provenance stream partitioning strategies: horizontal, vertical, and random for provenance emerging from data-intensive computations are also presented.

A provenance stream processing architecture is developed to apply the proposed parallel streaming algorithm on a stream of provenance arriving through a distributed log store. The solution is evaluated using Apache Kafka log store, Apache Flink stream processing

system, and the Komadu provenance capture service. Provenance identity, archival and reproducibility use a persistent ID (PID)-based approach.

Beth Plale, Ph.D.

David Leake, Ph.D.

Ryan Newton, Ph.D.

Judy Qiu, Ph.D.

Contents

Abstract i

List of Figures xiii

Chapter 1. Introduction 1

Chapter 2. Related Work 9

2.1. Data Provenance 9

2.2. Scientific Workflow Provenance 9

2.3. Provenance Querying 11

2.4. Big Data Provenance 11

2.5. Provenance Stream Analysis 13

Chapter 3. Background 15

3.1. Provenance 15

3.2. Big Data Processing Models 18

3.3. Distributed Log Storage 19

3.4. Persistent Identifier (PID) 21

Chapter 4. Related Previous Work 23

4.1. Komadu 23

4.2. Big Provenance In Data Lakes 33

Chapter 5. Provenance Stream Model 52

5.1. DICs and DIC Workflows 52

5.2. Backward and Forward Provenance 55

5.3. Provenance Streams 58

Chapter 6. Parallel Provenance Stream Processing 61

6.1. Big Provenance in DICs 61

6.2. Streaming solution for Big Provenance 65

6.3. Parallel-prov-stream Algorithm 70

6.4. Partitioning a Provenance Stream 74

6.5. Early Elimination Problem 79

6.6. PID based Provenance Identity in Data Lake 81

Chapter 7. Implementation and Evaluation 84

7.1. Parallel provenance stream implementation 84

7.2. Experimental Evaluation 90

Chapter 8. Conclusion and Future Work 101

Bibliography 106

Curriculum Vita

List of Figures

1.1 Reference Architecture 3

3.1 Each Kafka topic is divided into partitions (taken from Kafka docs) 19

3.2 Kafka consumer groups (taken from Kafka docs) 20

4.1 Sample Komadu provenance graph visualized using Cytoscape 27

4.2 Provenance Capture in SEAD VA using Komadu 28

4.3 Komadu Architecture 29

4.4 Data Lake Architecture 36

4.5 A high level data flow scenario in a Data Lake 41

4.6 Data Lineage 42

4.7 Data Lake use case 43

4.8 Cytoscape visualization of forward provenance for a single tweet 46

4.9 Cytoscape visualization of backward Provenance for one Spark output 47

4.10 Execution time with varying batch size 48

4.11 Hadoop Execution times for different scenarios 48

4.12 Spark Execution times for different scenarios 49

5.1 MapReduce DIC for hashtag counting 54

5.2 DIC workflow made up of six DICs 56

5.3 Provenance Graph for MapReduce job in 5.1 with Edge Types and Identifiers 56

5.4 Adding new Attributes to an Existing Node 59

6.1 Full provenance for MapReduce example illustrating forward provenance for first tweet and backward provenance for output $c : 2$. Direction of arrows follow W3C Prov convention for usage and generation. 62

6.2 How RAMP wraps Hadoop to capture provenance. Taken from [77] 64

6.3 Reduced backward and forward provenance 65

6.4 Stream processing solution for a DIC workflow 66

6.5 Stream processing solution for provenance generated by a streaming DIC 68

6.6 Possible derivation paths 69

6.7 Application of parallel stream processing on a partitioned stream of full provenance to produce a reduced provenance graph preserving backward and forward provenance 71

6.8 Reduction through the source vertex of a new edge 73

6.9 Partitioning provenance from a function execution 75

6.10 Horizontal partitioning 76

6.11 Vertical partitioning 77

6.12 A cluster partitioned based on locality 78

6.13 Early elimination example 79

6.14 Sliding window solution for the early elimination problem 80

6.15 Reduction with Grouped Usages and Generations 80

6.16 Extended streaming solution based on PIDs to handle provenance identity, reproducibility and archival 82

7.1 DIC workflow to categorize hash tags in Twitter data 85

7.2 Provenance stream processing architecture for a batch processing DIC workflow 86

7.3 Provenance stream processing architecture for a stream processing DIC. 88

7.4 Provenance from a streaming function is also considered as a parallel stream 89

7.5 Percentage accuracy of backward and forward provenance results against the percentage of out-of-order elements for “Grouping” and “Sliding Window” methods of our algorithm and Chen algorithm 93

7.6 Number of edges emitted by local reducers against the *local batch size* (in number of elements) for horizontal, vertical and random partitioning strategies 94

7.7 Speedup (parallelism = 1 time/parallelism = n time) of the system against increasing parallelism. Input data size is fixed at 10.1 GB and local batch size is 20000. 95

7.8 number of reductions (per minute) by the global reducer against the global batch size 99

7.9 Output reduced provenance throughput (MB/s) emitted by the global reducer against the global batch size 100

CHAPTER 1

Introduction

With the availability of the Internet infrastructure and the related technology, collecting data from various sources has become easier than ever before. These sources include but not limited to social media, click streams, sensors, IoT devices and server logs. As the volume of data continuously generated from these sources is exponentially increasing, Internet scale organizations routinely analyze such large volumes of data – also known as Big Data – for insight, decision making, and predictive forecasts. Computations used in this analysis, often belongs to the category of *large scale data-intensive computations* (DIC), uses parallel computing techniques and processing frameworks such as Hadoop [**16**], Spark [**108**], Flink [**26**], Storm [**99**] and Samza [**21**], designed for online and offline analysis.

Bringing data from different sources together and aggregating them can lead to useful insights which are not possible by processing them separately. Therefore scientists tend to store differently structured data from different sources in a single infrastructure – mostly built on a distributed file system like HDFS [**85**] or GFS [**46**] – and process them later based on different requirements. Modern DIC workflows which process such data often include multiple steps like data cleaning, joining data from different sources and applying processing algorithms. These steps in a given workflow may be performed using different Big Data processing frameworks at different points of time. Therefore the lifecycle of a data product which goes through a Big Data processing workflow becomes complicated.

Data Lakes [**48**] [**90**] [**96**] have emerged as a solution to Big Data management and analysis. The strength of the Data Lake is that it is considered to be schema-on-read where commitments to a particular schema are deferred to time of use [**91**]. Schema-on-read suggests that data are ingested in a raw form, then converted to a particular schema upon need. This applies to structured, semi-structured, and unstructured data; continuously arriving or not. Large scale Big Data processing frameworks are often integrated with Data Lakes and used to perform different transformation steps in data-intensive processing workflows. All outputs from executed transformations are written back to the shared storage layer so that future transformations can use those for further processing. The environment of the Data Lake, with multiple simultaneously, and long-running DICs, motivates the work presented in this thesis.

Such data-intensive processing environments are hard to manage [**9**] [**78**] as the data lifecycle inside them is so complicated. Given a data product, tracing its sources and finding all the processing steps applied on it is challenging. This kind of traceability is extremely important for many use cases. For example, imagine a situation where some sensitive information (like credit card information in user data) has been leaked into a data set and it has been used for multiple analyses. Now to find and clean all output data products which have been derived from the original data, there should be a good tracing mechanism implemented.

In our initial work [**91**] [**92**] we assessed the traceability issues in such data-intensive computations and introduced a data provenance based solution to overcome those. Traditionally data provenance is used to trace the lineage of a data product from the origin

including all processing steps and people involved. Based on this concept, we came up with a reference architecture to collect, manage and analyze provenance data within a Data Lake and evaluated it through a prototype implementation.

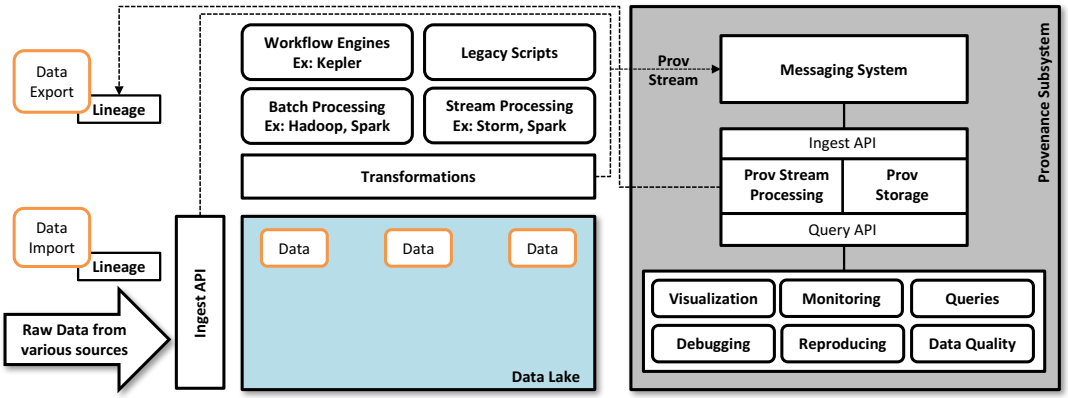


FIGURE 1.1. Reference Architecture

Figure 1.1 shows the reference architecture that was presented in our initial work. Provenance events from ingest APIs and from different transformations are streamed into the central provenance subsystem. Here we focus on fine-grained provenance events through which even the simplest processing steps in the transformation can be traced. Therefore, the amount of provenance data collected in the provenance store can be large. In the prototype implementation, we used Komadu [**93**] provenance management system to persist and query provenance. Through our experiments, we showed that the size of fine-grained provenance could be several times larger than the input data size. There is considerable evidence accumulated through years of provenance research to support that claim [**53**] [**103**]. Managing such large volumes of provenance data and providing useful analysis on them are identified as challenges [**103**] in Big Data provenance and named as the "Big Provenance" problem in the literature [**47**].

Storing and managing Big Provenance for later analysis may not be feasible for most data-intensive applications due to high resource consumption. If the collected provenance is multiple times larger than the input data for a certain DIC workflow, the storage layer should be expanded just to store provenance and more compute nodes should be allocated for provenance processing. Querying stored Big Provenance is also problematic [**50**] [**38**] due to high resource consumption for graph traversal queries. Even if the resources are available, analysis results will be delayed due to offline processing. Another point to consider is that, not all provenance is equally valuable: provenance for intermediate data products in a data-intensive computation is not directly used most of the time. Therefore, retaining full provenance could be a waste of resources in most cases.

We focus on streams of provenance data. A data stream is a sequence of data elements arriving over time. A stream processing abstraction is applied in cases where analysis occurs in real-time on a portion of the stream, often requiring only one pass over the data. In this thesis, we apply the stream processing abstraction to provenance emerging in real-time from a data-intensive computation. This minimizes the cost of retaining full provenance while providing a sufficient level of analysis in (near) real-time. Our approach is to use stateful, one-pass parallel stream processing to summarize a full provenance stream on-the-fly by preserving backward provenance and forward provenance. It does this through maintaining state within each parallel stream consumer. Intuitively, *backward provenance* begins with an output and traces backwards in time to the subset of input data on which the output depends. *Forward provenance* begins with an input and traces forward in time to the subset of output data elements derived by it.

There are well known provenance representations like W3C PROV [**67**] and OPM [**69**]. Most of these representations model provenance information as a directed acyclic graph. A vertex in a W3C PROV graph can represent an activity, entity or agent. An edge between two vertices is always a provenance relationship between them. For example, if the entity A was generated by the activity B, the edge between them represents the *wasGeneratedBy* relationship. We define a "Provenance Stream" in the coming sections so that each individual provenance element in the stream either represents an edge in a provenance graph or adds attributes to an existing vertex. According to our definition, a provenance stream is always a graph stream.

Computing *backward provenance* for outputs and *forward provenance* for inputs are the mostly used advantages of data provenance. *Backward provenance* is the set of input data items on which a given output data item depends. *Forward provenance* is the set of output data items derived by a given input data item. Traditionally, both backward and forward provenance on stored static provenance graphs are calculated using techniques such as recursively traversing through the graph [**50**] and maintaining transitive closure tables [**38**]. Such techniques do not scale for Big Provenance due to high compute and storage overhead. So we develop *parallel-prov-stream*, a parallel stream algorithm for reducing full provenance from a data-intensive computation on-the-fly by preserving backward and forward provenance that is scalable, order independent and one-pass only. This avoids the need for full provenance storage and provides (near) real-time results.

The stream processing approach is framed in the context of a DIC workflow where multiple DICs run concurrent with each other, are long lived, and are distributed. Provenance is streamed to a single, standalone provenance stream processing system that is itself distributed. There are two main modes of execution in data-intensive computations: batch mode and streaming mode. A batch computation consumes a stored static dataset and terminates within a finite amount of time. A streaming computation consumes a stream of data and may run forever. We discuss our provenance streaming solution for provenance generated by both the categories. From a provenance perspective, main difference between the two modes is that a provenance stream from a batch computation is always terminating and a provenance stream from a streaming computation could be non-terminating. Therefore we adapt our streaming solution for both terminating and non-terminating provenance streams.

The final framing of the approach is its implicit treatment of *provenance identity*. The provenance system is built based on a log store abstraction that supports "topics" to which provenance events are published. Topics, and their uniqueness, is used to guarantee that provenance events are associated with the correct DIC from which they were generated. We assume that when a new DIC begins, it is configured to publish provenance to a unique topic ID in the distributed log store. Furthermore, we discuss provenance identitiy, archival of reduced provenance graphs and reproducibility of provenance stream processing runs using a Persistent Identifier (PID) [**106**] [**7**] based extended architecture.

Closest to our research is Chen and Plale [**28**] where a dependency matrix is computed across input parameters and variable values from a stream of data provenance from an

Agent-Based model. Our work here both extends and complements Chen and Plale through application of provenance stream processing to large scale DICs. In the context of agent-based simulations they assume that the stream of provenance is ordered based on generation time when it arrives at the provenance stream processing system. This assumption may not hold for data-intensive computations and we lift it by handling out-of-order provenance events. In addition to that, they execute their algorithm using a single stream consumer which is not good enough to handle higher provenance event rates. Our algorithm is parallel and scalable for multiple consumers reading from the same stream of provenance.

Following are the main contributions of this dissertation.

- A provenance model which includes formal definitions for provenance stream, forward provenance and backward provenance in the context of data-intensive computations. Here we start from the function level provenance in a DIC and extend our definitions up to DIC workflows. For the purposes of this thesis, we use the relationships defined in W3C PROV specification.
- A stateful, one-pass, parallel stream processing algorithm to summarize a full provenance stream on-the-fly by preserving backward provenance and forward provenance. The algorithm is resilient to provenance events arriving out-of-order within some time delta. We simulated an out-of-order provenance stream to measure the accuracy of the output under varying out-of-order levels. Multiple provenance stream partitioning strategies: horizontal, vertical, and random, for data-intensive computations are also presented. We evaluate performance and accuracy of the results under each partitioning strategy.

- A provenance stream processing architecture which applies the proposed parallel streaming algorithm on a stream of provenance arriving through a distributed log store. The streaming solution is evaluated using an implementation based on Apache Kafka log store, Apache Flink stream processing system and Komadu provenance repository. Provenance identity, archival and reproducibility are also discussed using a PID based approach.

The remainder of this thesis is organized as follows. Chapter 2 provides a detailed discussion of related work. Chapter 3 presents high-level descriptions of the concepts used in the thesis. In Chapter 4 we discuss our previous works on Komadu provenance repository and provenance solution for Data Lakes. This provides a detailed explanation on Data Lakes and lays the foundation for the work we present in this thesis. Chapter 5 presents our provenance stream model which includes the formal definitions. Then in Chapter 6 we discuss our parallel provenance stream processing algorithm and provenance stream partitioning strategies in detail. Our implementation and evaluation results are presented in Chapter 7 and the thesis concludes with future work in Chapter 8.

CHAPTER 2

Related Work

2.1. Data Provenance

Research on data provenance was initiated from the database community [**107**] [**25**] [**24**] where the main focus is to derive the history of a given tuple or a data object in a database. Interest for provenance in eScience community increased in early 2000s as provenance in scientific workflws [**111**] [**11**] [**75**] was identified as a crucial component. A number of early provenance collection systems such as Vistrails [**82**], Karma [**88**], and PASS [**72**] took part in the first provenance challenge [**71**] which was held in 2006 to understand the common types of provenance queries that should be supported by a provenance system. Second provenance challenge was held shortly after that focusing on understanding the issues in interoperability in provenance and the OPM [**69**] standard was the outcome. As researchers identified some limitations in OPM while trying to apply it in different domains, the third provenance challenge [**86**] was held and the W3C PROV [**67**] specification was defined.

2.2. Scientific Workflow Provenance

Provenance capture, query, and visualization on traditional scientific workflows is a well-studied area [**87**]. Most of these studies focus on scientific workflows running on grids and captures coarse-grained provenance. Chimera [**42**] is a virtual data system which supports provenance. It provides derivation traces for scientific data. MyGrid [**111**] is a

provenance enabled middleware framework for SOA based workflows. There are similar studies such as [**43**] and [**74**] which focus on provenance in application-specific scientific workflows. In addition to the above systems, there are well known scientific workflow execution frameworks such as Kepler [**11**], Taverna [**75**], VisTrails [**82**] which captures provenance automatically from the workflow being executed. Most of the time, such workflows include a number of services for different tasks. Workflow provenance captures service invocations and all used and generated data products from each service.

Karma [**88**] is one of the first attempts to build a general purpose provenance collection framework for scientific workflows which uses a standard provenance representation language. It stores provenance events in a relational database and generates provenance graphs in OPM standard. Komadu [**93**] is the successor of Karma which supports W3C PROV [**67**] as the provenance representation language. Both Karma and Komadu can be executed as a standalone provenance repository which can be used to collect provenance events from distributed applications. They provide provenance query APIs and visualization tools as well for analyzing provenance graphs.

There are few systems like PASS [**72**] and SPADE [**44**] which collects provenance at the operating system level. PASS captures provenance at the file system level and stores with each file. SPADE captures all system calls including their inputs and outputs which leads to large volumes of low-level fine-grained provenance. SPADEv2 [**45**] is an improved version specially designed to support distributed applications. As it could produce large volumes of provenance, they argue against central provenance storage systems like Karma and locally stores provenance in each node in a distributed environment. However, this

leads to performance issues in query time as provenance stored in different nodes should be integrated. This kind of OS-level provenance collection is too low level and hard to manage in modern Big Data processing applications.

2.3. Provenance Querying

PASS introduced a new query language [**52**] which is based on paths in a provenance graph. It supports regular expression like queries as well to find paths which include certain inputs, outputs or functions. They argue against provenance storage mechanisms like relational databases, XML, RDF etc. and propose a semi-structured data model for provenance for efficient querying. QLP [**12**] is a declarative provenance query language which also supports regular expression like queries. It provides the advantage of executing queries on OPM as well. They use transitive closure tables on each node in a provenance graph to improve the efficiency of graph traversal queries. SPADEv2 uses a technique called provenance sketches [**61**] to improve query efficiency in their decentralized provenance storage architecture. They argue that recursive computation and transitive closures are less efficient in querying large provenance graphs. Heinis, T. et al. [**51**] also supports that argument and propose a different approach called "interval encoding" to improve query efficiency in graph traversal queries. They transform provenance graphs into trees by duplicating nodes to apply interval encoding.

2.4. Big Data Provenance

When moving from traditional scientific workflow systems to modern Big Data processing frameworks, above systems and techniques suffer number of challenges in storage,

scalability, querying etc. [**35**] [**103**] [**47**] [**95**]. Glavic, B. et al. [**47**] highlight the importance of provenance in Big Data for purposes like performance analysis, debugging and reproducibility and name it *Big Provenance* as the volumes of provenance itself can grow rapidly. Wang, J. et al. [**103**] identify the challenges and opportunities in *Big Provenance*, focusing on distributed data-parallel computations. They come up with a reference architecture for Big Data provenance and identify (1) volume of provenance, (2) provenance integration and (3) collection overhead as the main challenges in *Big Provenance*. In this thesis, we address a few of those challenges.

There are few studies in the recent past which focus on capturing provenance in Big Data processing frameworks like Apache Hadoop and Spark. Wang J. et al. [**104**] [**34**] present a way of capturing provenance in MapReduce workflows by integrating Hadoop into Kepler and using provenance capabilities of Kepler. RAMP [**77**] [**53**] extends Hadoop to capture provenance by propagating input identifiers through the computation. They have built wrappers for Hadoop which automatically record provenance when a job is executed. HadoopProv [**5**] modifies Hadoop to reduce provenance capturing overhead. Both RAMP and HadoopProv capture fine-grained provenance which includes intermediate data as well. They are capable of supporting both backward and forward provenance between inputs and outputs. Titian [**54**] is a modified version of Spark which automatically captures provenance from any function applied on a dataset. They have made the interactive Spark shell provenance-aware so that lineage can be accessed for any data object as soon as a function is executed. A new provenance enabled RDD interface called LineageRDD has been added to the API so that lineage can be accessed for any RDD. This helps in debugging

Spark computations quickly and easily. Sansrimahachai, W. et al. [**79**] presents a way of capturing provenance from stream processing systems which is useful in real-time tracing of data objects and validating processing steps.

There are few Big Data processing workflow management tools which support chained computations. Apache Falcon [**14**] framework manages the data lifecycle in Hadoop Big Data stack. Falcon supports creating data processing pipelines by connecting Hadoop based processing systems. It captures lineage of data while the pipeline is being executed and visualizes the lineage graph at the end. Apache Nifi [**18**] is another data flow tool which captures lineage while moving data among systems. However, neither tool captures detailed provenance information within transformation steps. Therefore they do not support detailed analysis over provenance. In our previous work [**91**] we presented techniques to capture and fine-grained provenance from data-intensive computations and integrate them across multiple transformations performed by different frameworks. We built our argument around a Data Lake environment where unique identifiers for data products across computations are assumed. Through the experiments, we saw that the amount of provenance collected for could be multiple times larger than the original data size and that becomes a problem when managing and querying provenance.

2.5. Provenance Stream Analysis

As a provenance stream is always a graph stream, graph streaming techniques are applicable in provenance stream analysis. Only a limited number of studies on graph streaming is found in the literature. Applying static graph analysis methods on graph streams is considered a difficult problem especially when the graph is directed [**64**]. Algorithms for

problems like triangle count [**22**] and page rank [**81**] have been studied. There are few studies on graph stream clustering [**27**] [**2**] and mining [**33**] as well. McConville, R. et al. [**63**] recently presented a vertex clustering mechanism for graph streams while Manzoor, E. A. et al. [**62**] presented an anomaly detection mechanism.

Capturing provenance from streaming data is another related area. Vijayakumar, N. et al. [**101**] [**102**] present techniques to capture provenance among data streams. Fine-grained provenance capturing methods for streaming data have also been studied [**65**] [**79**] [**80**]. Sansrimahachai, et al. [**79**] processes provenance as a stream and apply queries on top of it.

To best of our knowledge, the only attempt in the literature to use stream processing techniques for analyzing a stream of provenance is made by Chen, P. et al. [**28**] where the authors compute a dependency matrix between the input parameters and the variable values in an Agent-Based simulation using a stream of provenance. However, there are multiple drawbacks in their work. Their definition of a provenance stream only allows derivation relationships and that limits its applicability in most applications. They assume that the stream of provenance is ordered based on generation time when it arrives at the provenance stream processing system. This assumption is far from reality when applied in a distributed data-intensive computation. There can be many reasons like network delays, lost packets, re-transmits etc. which breaks the creation order in provenance events. In addition to that, they execute their algorithm using a single stream consumer which is not good enough to handle higher provenance event rates.

CHAPTER 3

Background

3.1. Provenance

Provenance is information about entities, activities, and people involved in producing a piece of data. Provenance is also considered as a special type of metadata associated with data. A provenance record attached with a data product improves the value of the data as it helps in many ways. Assessing data quality, reliability, and trustworthiness, tracing the sources of data, reproducing and debugging the computations used to derive data are few advantages of provenance.

Provenance is often used in eScience to improve the reusability of scientific data. Research on provenance in eScience has been there for more than a decade. Provenance capture, storage, visualization, and querying are few areas which have been explored so far. Often provenance is captured from scientific workflows by using methods like log monitoring and instrumentation. Captured provenance is mostly stored in provenance management systems like Karma [**88**] and Komadu [**93**]. Visualizations and queries are also supported by such systems to make provenance information useful. Provenance representation models like OPM [**69**] and W3C PROV [**67**] have been introduced to increase the interoperability of provenance data collected from different systems.

Open Provenance Model (OPM)

OPM [**69**] was the first data model introduced for provenance in order to achieve interoperability among systems. OPM represents provenance as a directed acyclic graph in which nodes represent artifacts, processes and agents defined as follows.

Artifact: Immutable piece of state, which may have a physical embodiment in a physical object or a digital representation in a computer system.

Process: Action or series of actions performed on or caused by artifacts, and resulting in new artifacts.

Agent: Contextual entity acting as a catalyst of a process, enabling, facilitating, controlling, or affecting its execution.

Edges between nodes are used to represent relationships between the above elements. OPM defines five types of relationships.

- *used*: Artifact used by a Process
- *wasGeneratedBy*: Artifact generated by a Process
- *wasTriggeredBy*: Process triggered by Process
- *wasDerivedFrom*: Artifact derived from Artifact
- *wasControlledBy*: Process controlled by Agent

With the usage in different applications, it was understood that OPM is limited in some areas. It does not define any Agent-to-Artifact or Agent-to-Agent relationships. In addition to that OPM does no allow different types of derivations between Artifacts.

W3C PROV

W3C PROV [**67**], [**68**] was introduced later to address the limitations of OPM and to define few more new concepts which are useful in recording provenance. The three main

elements defined in OPM has also been renamed and redefined as follows. However, they still carry similar meanings as in OPM.

Activity: An activity is something that occurs over a period of time and acts upon or with entities; it may include consuming, processing, transforming, modifying, relocating, using, or generating entities.

Entity: An entity is a physical, digital, conceptual, or other kind of thing with some fixed aspects; entities may be real or imaginary.

Agent: An agent is something that bears some form of responsibility for an activity taking place, for the existence of an entity, or for another agent's activity.

PROV model defines a rich set of relationships between the elements defined above.

- Entity-Entity: *wasDerivedFrom*, *alternateOf*, *specializationOf*, *hadMember*
- Entity-Activity: *wasGeneratedBy*, *used*, *wasInvalidatedBy*, *wasStartedBy*, *wasEndedBy*
- Entity-Agent: *wasAttributedTo*
- Activity-Agent: *WasAssociatedWith*
- Activity-Activity: *wasInformedBy*
- Agent-Agent: *actedOnBehalfOf*

There are many improvements in these relationships compared to OPM. For example, *wasDerivedFrom* relationship in PROV defines multiple subcategories; Revision, Quotation, Primary Sources. Those can be used to describe exact type of derivation. In addition to that, PROV defines a model to represent collections of entities through the *hadMember* relationship. Connecting Entities with Agents to represent attribution is also possible using the *wasAttributedTo* relationship. With such improvements over OPM, W3C PROV has

become the most used and supported provenance standard in the modern day provenance applications.

3.2. Big Data Processing Models

Batch Processing

Processing large sets of stored data using parallel computing techniques to achieve offline results is referred to as "Batch Processing". The most notable feature of this kind of computations is the availability of full data. Processing algorithms are run on the full data set to generate results. Batch processing is mostly used to run complex algorithms (Ex: Machine Learning) and often take hours to complete. Time taken to produce results is not an issue and the focus is on the throughput. There are multiple batch processing techniques such as MapReduce [**36**] and Spark programming model [**108**] and frameworks such as Hadoop MapReduce [**16**], Spark [**108**] and Flink [**26**]. All these frameworks are highly scalable and fault-tolerant.

Stream Processing

A data stream is a sequence of data elements arriving over time. Stream Processing is used for real-time (or near real-time) analysis on a data stream considering only a time-based or rate-based window of the stream at a given point of time. Often, stream processing algorithms use only one pass over data. Due to such reasons, stream processing is used for simpler analysis compared to Batch Processing. Modern stream processing frameworks such as Storm [**99**], Twitter Heron [**59**], Samza [**21**], Spark Streaming [**109**] and Flink [**26**] focus on fault-tolerant and distributed processing of streaming data to achieve low latency

and high throughput. There are two widely used stream processing techniques: one-at-a-time and micro-batch. Each stream data element is processed immediately upon arrival in one-at-a-time processing model which is also considered as true stream processing, used by Storm and Flink. Sub-second latency is targeted in one-at-a-time processing model. Few stream data elements are batched together for processing in micro-batch processing model which is a micro version of batch processing, used by Spark Streaming. Micro-batch processing targets second level latency while providing higher throughput compared to one-at-a-time model. Stateful computations are difficult to handle in distributed stream processing. Flink and Samza support stateful computations up to some extent.

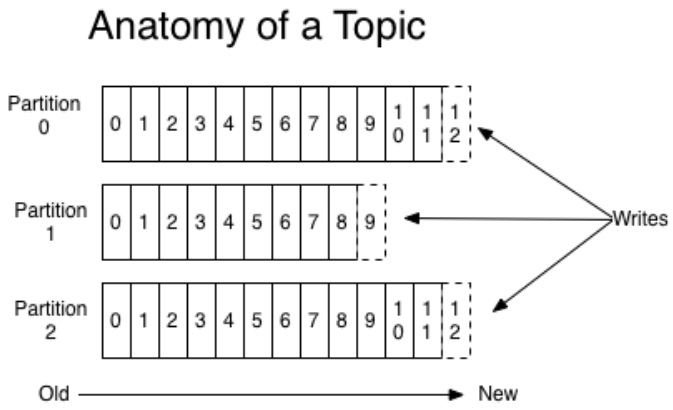


FIGURE 3.1. Each Kafka topic is divided into partitions (taken from Kafka docs)

3.3. Distributed Log Storage

A distributed log storage is generally a system designed for short-term persistence of streaming data. The main goal behind such a system is to achieve low latency and high throughput for fast moving streaming data. Two widely used distributed log stores are Apache Kafka [**58**] and Amazon Kinesis [**83**]. We use Kafka in this thesis.

Apache Kafka

Apache Kafka is a distributed streaming platform originally developed by LinkedIn for collecting and delivering high volumes of log data with low latency. The concept of *topic* in Kafka is based on the log abstraction where a *log* is an append-only sequence of records with new records added to the end. A topic is a category or feed name to which records in a stream are published. There can be multiple consumers for each topic who read records from it. As shown in Figure 3.1, each topic is divided into partitions to achieve scalability and high throughput. A *partition* is an ordered, immutable sequence of records that is continually appended to. Each record within a partition is uniquely identified by an offset or a sequence number. When multiple consumers are consuming records from the same partition, they may consume at their own rates and each of them maintains an offset within the partition. Partitions are replicated for fault-tolerance.

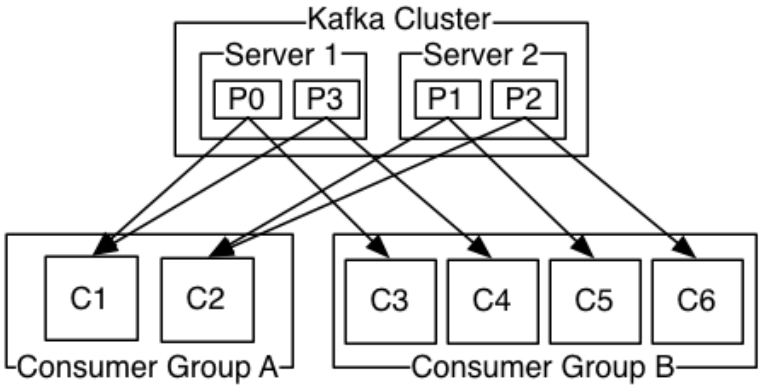


FIGURE 3.2. Kafka consumer groups (taken from Kafka docs)

Kafka records are always retained for a configurable retention period irrespective of whether they are consumed or not. Therefore, the consumers have the flexibility to read from any offset within the retention period. The concept of consumer groups can be used

for parallel consumption of records which leads to higher throughput and fault-tolerance. Figure 3.2 shows how consumers in a group can consume records from partitions in a topic. Each record can be consumed by only one consumer within a group. Partitions in a topic are distributed among the consumers in a group so that all records in a given partition are consumed by only one consumer.

3.4. Persistent Identifier (PID)

A Persistent Identifier is used to uniquely identify a digital data product which can be a file, a collection or any type of digital object. It guarantees a persistent one-to-one relationship between the identifier and the data product. Using the persistent identifier, a data product and its metadata can be retrieved at any point in time. It has become an important data practice to assign persistent identifiers for digital objects to make sure they are preserved for long-term usage, especially in use cases such as Data publishing [**73**], reproducibility [**55**] [**31**] and archival. DOIs [**37**] is the mostly used persistent identifier system for digital objects like publications and articles. Handle system [**49**] and Archive Resource Key [**98**] are among other widely used systems.

Most of the above systems are designed targeting humans as the end user of the persistent identifier. When an identifier is resolved, the user is redirected to a landing page through which the digital object and its metadata can be retrieved. However, when used by software systems, above systems suffer from issues like weak interoperability and inconsistent protocols for getting from PID to the data object. The Persistent Identifier (PID) System [**106**] [**57**] was designed to address this problem. It is intended to be used with software systems so that millions of identifiers can be resolved at internet speed. PID

architecture is based on the Handle System and RDA Data Type Registry [**6**]. A PID consists of a kernel which includes minimal but sufficient metadata about the digital object.

CHAPTER 4

Related Previous Work

In this chapter we present our two previous works which laid the foundation for the provenance stream processing solution presented in this thesis. First Komadu, a W3C Prov based provenance repository which supports provenance ingest and query capabilities. Second, an architecture and a prototype implementation for provenance collection and management in a Data Lake environment to address the accessibility and traciability concerns. Challenges identified in storing and analyzing Big Provenance in a Data Lake provided the motivation for provenance stream processing work.

4.1. Komadu

Data provenance is information about the entities, activities and agents who have effected some type of transformation on a data product through the its lifecycle. Data provenance captured from scientific applications is a critical precursor to data sharing and reuse. For researchers wanting to repurpose and reuse data, it is a source of information about the lineage and attribution of the data and this is needed in order to establish trust in a data set. Data provenance has been shown useful in results validation, failure tracing, and reproducibility. The Komadu provenance capture system is standalone, meaning it is not coupled to or dependent upon any database management system, repository, or scientific workflow system. It provides an ingest API through which provenance notifications are fed

into the system at high speeds, and a query API through which provenance information can be queried. The data model is both event oriented and graph oriented, in that graphs are pieced together in Komadu based on the events received from the environment.

Komadu has its roots in the Karma [**88**] provenance capture system, an earlier version that complied with the OPM [**69**] community standard both for defining the type of provenance notifications that the system accepted, and for defining the format of the results. Komadu, on the other hand, supports the W3C PROV specification [**67**] which provides far richer types of relationships and has a more formal model for handling time than does OPM. Karma was additionally limited by assuming that every notification belonging to the same external activity shared a common global identifier that is shared across all components (services, methods etc.) of the external environment. This limitation was found to be severe in applications where provenance is not only captured at the application level, but also at in the larger environment where the application runs. Take for instance a distributed application running on Twister [**40**] using PlanetLab [**32**] as the underlying infrastructure; it is highly limiting to expect provenance events generated from the application, from Twister, and from PlanetLab to all have shared knowledge about any single global identifier. This limitation derives from Karma's early days where it tracked provenance for applications running within a single workflow system. Additionally, a researcher may be interested in tracking lineage starting from some data product or agent. Such scenarios are not supported by Karma.

Komadu is a complete redesign and reimplementation of Karma that supports new features while addressing the above mentioned limitations of Karma. The main contributions of Komadu are as follows.

(1) PROV Support: Komadu is completely W3C PROV specification compliant.

(2) Beyond Workflow: Komadu client API is simple and designed according to standards defined in the specification. Therefore, Komadu is not restricted for scientific workflows and can be used for capturing provenance from any kind of application.

(3) Context-free: Unlike Karma, the graph generation algorithm used in Komadu does not depend on any global context identifier. This makes it possible to collect provenance from disparate and unrelated pieces of infrastructure and application. It of course introduces the challenge to be handled within Komadu of stitching together graphs based on events that are not easily identifiable as being causally related. Komadu is backward compatible with Karma through graph generation that uses global context identifiers. A new user is advised to use the more convenient context-less mechanism.

(4) Multiple Perspectives: In addition to generating provenance from the perspective of an activity, Komadu is capable of generating provenance graphs starting from data products and agents as well.

Provenance is generated through first instrumenting an application, a tool, or system middleware directly; or by processing log files after executing the application. Komadu has support for both generation mechanisms. Queries can be executed any time once the

notifications are ingested. Komadu comes with an Ingest API and a Query API, both of which are exposed as a Web Service and a Messaging service.

A tool to enable the visualization of provenance graphs is included in the package as this aids the researcher in making sense of what can be gigabytes and terabytes of provenance. When the provenance is voluminous, which it can easily be, it is hard to extract important information through any other means. Komadu comes with a simple command line tool that converts the generated XML graph into a CSV file. This CSV file can be imported into most of the visualization tools. We use Cytoscape **[84]** to visualize provenance generated from Komadu (and Karma before it) and have written a provenance specific plugin for Cytoscape. Figure 4.1 shows a sample provenance graph generated by Komadu and visualized using Cytoscape. This graph includes dummy Activities, Entities and Agents generated by one of the Komadu integration test cases. An Agent (Agent˙16) invokes a service (Activity˙142). Activity˙142 uses a collection of files (Collection˙661) in the generation of an output file (File˙660). Service (Activity˙142) additionally invokes service (Activity˙143); the latter reads input file, File˙663, and generates output file, File˙659. All elements and relationships contain additional attributes and those are displayed in a separate window when a specific part of the graph is selected.

4.1.1. Motivation. An example of Komadu use in a scientific setting is in the Sustainable Environment Actionable Data (SEAD) **[73]** DataNet project. SEAD is developing tools for data management in the long tail of science. The scientific community in the US is recognizing the need for increased availability of the data products of their research, but the mechanisms for submitting data sets to public repositories still involve considerable manual

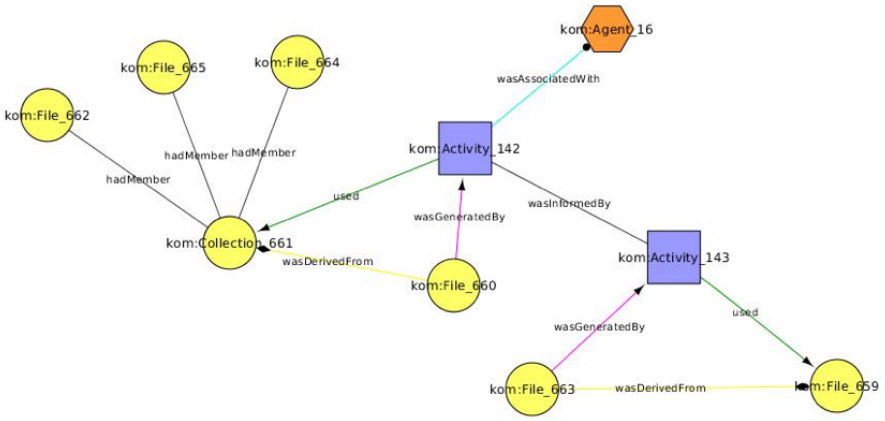


FIGURE 4.1. Sample Komadu provenance graph visualized using Cytoscape

effort. SEAD tools are attempting to reduce the manual barrier to submission of data from scientists in the long tail, those that generate small but highly voluminous data sets. SEAD had adopted the notion of the Research Object (RO) [**23**] as the unit of preservation, and uses Komadu to track the lifecycle of the RO through derivation, revision, and reuse.

The use of Komadu in SEAD is captured in Figure 4.2. A sustainability science project sets up a shared Project Space for its work. When a collection is ready for publication to a public repository, it is published, whereupon the SEAD Virtual Archive (VA) picks it up, and curates it. A BagIt service inside VA pulls collection metadata and data from Project Spaces and generates additional metadata. An ingest workflow is then invoked, carrying out functions on the ingest package. The provenance of the activity is published into Komadu. Once the data collection is ingested, the data curator who often works in an academic library, can edit related metadata using the VA user interface. Provenance events of the curation actions are also sent to Komadu. Finally, when the curation is complete, the data curator publishes the collection into an appropriate Institutional Repository.

Provenance events related to publish workflow are also captured. VA components like BagIt, Workflows and Registry are distributed components that are connected using web service interfaces. All those components push provenance events into Komadu using the web service interface exposed by Komadu. The SEAD VA allows scientists to search for published data collections. When a particular collection is selected, a provenance graph related to that collection is visualized in a separate provenance window. This provenance graph contains the relationships between collections and also provenance information inside each collection.

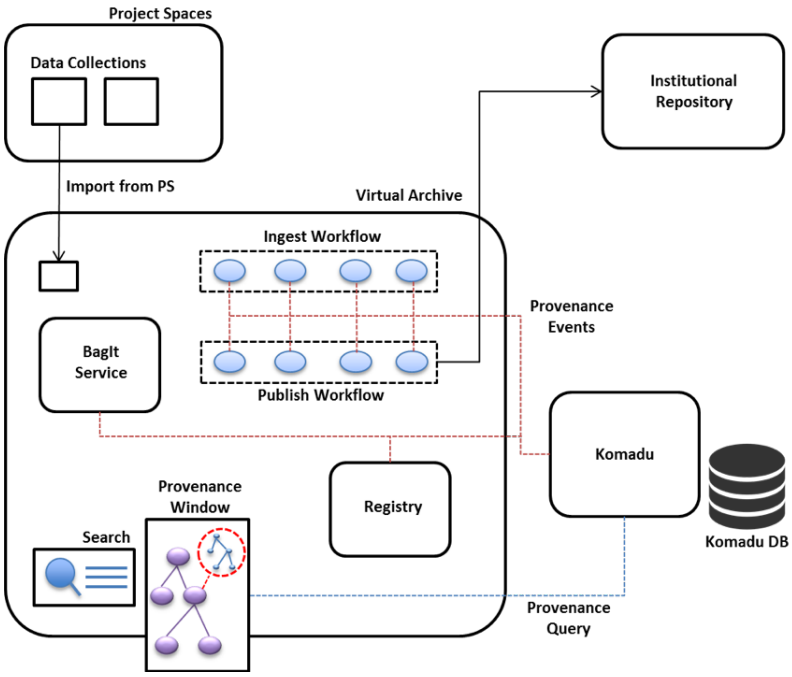


FIGURE 4.2. Provenance Capture in SEAD VA using Komadu

4.1.2. Implementation and architecture. Figure 4.3 shows the high level architecture of Komadu. Komadu can be run as a Web Service hosted on Apache Axis2 [**13**] or as a standalone server that listens to a RabbitMQ [**100**] message queue. In both cases, a client

has to be created to send messages into Komadu. A research programmer instruments a researchers application, services, and tools to ingest provenance notifications into Komadu on the fly or can use a log processing script that parses execution logs for provenance information after the execution of the application. Queries can be issued to Komadu to generate provenance graphs and retrieve provenance information related to Activities, Entities or Agents. Both ingest and query APIs are exposed through Web Services and RabbitMQ channels, the RabbitMQ channels are set up and configured by the research programmer.

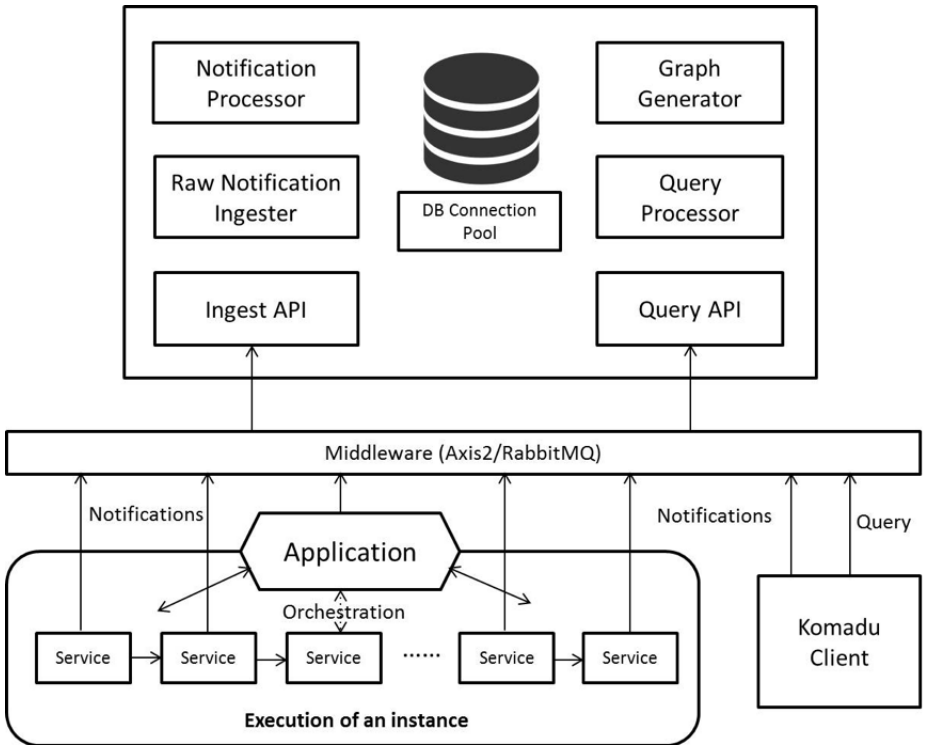


FIGURE 4.3. Komadu Architecture

Ingest API: The Ingest API is used to send provenance notifications into Komadu during provenance collection time. XML notifications must be compliant with the Komadu

XML Schema. For example, if a service A invokes a service B using some parameters, the notification will contain the identifiers of Service A and B and the required parameters.

Query API: The Query API is used to issue queries to the Komadu server anytime after the provenance is captured. Queries can be issued to retrieve information about certain Activities, Entities etc. or to get the generated provenance graphs. Queries also must be compliant with Komadu XML Schema.

Database: Komadu uses a MySQL database to store all incoming notifications, processed components, their relationships and generated provenance graphs. A connection pool is used to create and efficiently manage database connections under high data rates.

Raw Notification Ingester: This component is responsible for ingesting incoming XML raw notifications into the database as quickly as possible. Once the message is ingested, the incoming thread is immediately returned to make the server more responsive under high loads. Raw notifications are processed by a separate component running asynchronously.

Asynchronous Raw Notification Processor: This component is responsible for processing raw notifications in the database asynchronously. It consists of a pool of threads that run periodically and check whether there are unprocessed notifications left in the database. If such notifications are found, those are processed and split into Activities, Entities and Agents and stored back into the Komadu database. Once processed, raw notifications are marked as processed in the database. Each notification comes as a relationship between two elements (Ex: Activity-Activity, Activity-Entity etc.) and contains element identifiers. Raw notification processor creates elements in the database if those do not already exist and adds the relationship between those elements.

Query Processor: query processor is responsible for handling all incoming queries. There are two types of queries: (1) queries that request specific information about a certain Activity, Entity or Agent, (2) graph queries. The former are easy to handle and the Query Processor directly accesses the database to respond to such queries. For the latter, the Graph Generator is invoked with the relevant start node identifier sent by the client.

Graph Generator: The graph generator generates provenance graphs for incoming node identifiers. It starts by creating the start node with the incoming Activity, Entity or Agent identifier, then continues to add connected nodes transitively into the graph until there are no connected nodes left. The graph generation process uses a depth first approach and a stack of unexpanded nodes. Graph Generator uses caching to improve performance where the cache interval is configurable by the system administrator. When a new graph is created, it is cached in the database and if the same graph is requested again within the cache interval, cached graph is returned. If the cached version is expired, a new graph is created and the cache is updated.

RabbitMQ Messaging Channel: RabbitMQ is an eventing system supporting asynchronous communication. A publisher publishes messages to a middleware, often called a broker; one or more subscribers subscribe to channels on which the publishers place notifications. Komadu uses RabbitMQ as a messaging broker to receive provenance notifications and send responses to the incoming queries. RabbitMQ provides a persistent and reliable store for messages.

Axis2 Web Service Channel: When Komadu is deployed as a Web Service on top of Axis2, a service client can be used to communicate with the server using the SOAP messaging protocol. A service client can be easily generated using the Komadu WSDL file.

Komadu is implemented completely in Java programming language and it uses MySQL as the backend database. Komadu uses the Prov Toolbox [14] library in the process of generating provenance graphs. Komadu API is clearly defined as an XML schema and it is exposed on both Web Services and Messaging channels.

4.1.3. Application of Komadu. The main use of Komadu is for tracking lineage of data generated and used in scientific research. As a standalone system, provenance can be aggregated from tools, services, applications, and middleware. The generated provenance traces have been shown as useful to reproduce the workflow execution and to trace failures.

As mentioned previously, Komadu is the successor of the Karma provenance capture system. One of the drawbacks of Karma is that its APIs are tightly coupled with workflows. Almost all operations in Karma client API are using workflow related terms. Therefore, it is hard to use Karma to collect provenance data in other settings. One of the main goals of Komadu was to overcome this limitation. Komadu APIs are designed using the generic definitions used in the W3C Prov [**67**] specification. Therefore, Komadu can be easily used by any kind of application where provenance data has to be collected.

Another area where Komadu is useful is research data preservation repositories like SEAD. Actions are taken on research/data objects that reside in long term repositories. These actions could affect the object and thus should be part of the provenance record. For example, once a dataset is submitted into a repository, number of data curators can

edit metadata or transform the dataset into different formats. Provenance can be used to distinguish changes to a research object that can be constituted a revision (e.g., by same author, to correct error) from those that should be viewed as a derivation (e.g. subset of data object used for another purpose). This can be accomplished by integrating Komadu with the preservation repository.

A single standalone provenance tool like Komadu can serve as an aggregator of provenance from multiple sources. There is nothing preventing Komadu from representing other data manipulation processes that occur in industry or government. Finally, it can be useful for big data processing frameworks like Apache Hadoop [**16**] and Apache Storm [**99**] in certain applications to track lineage of produced data products. Users can integrate Komadu in their application specific code (Ex: In Hadoop mapper or reducer) to collect provenance data.

4.2. Big Provenance In Data Lakes

Today's internet scale organizations routinely analyze Big Data from various sources for decision making and forecasting. These sources (e.g., clickstream, sensor data, IoT devices, social media, server logs, databases) are external to an organization and internal. Much of it is continuously being generated (social media, sensor data). Depending on the source, data can be structured, semi-structured or unstructured. Traditional data warehouse is proving to be too inflexible and limited [**90**] when it comes to storing different types of data from numerous sources mainly because of the schemas enforced at ingest time. Conversion of data to fit a particular ingest schema may lose considerable amount of information even before the start of analysis.

A new response to the limits of the data warehouse is the Data Lake [**8, 90, 97**]. The Data Lake is considered as schema-on-read where commitments to a particular schema are deferred to time of use. Schema-on-read suggests that data are ingested in a raw form, then converted to a particular schema when needed to carry out analysis. This applies to structured, semi-structured, and unstructured data; continuously arriving or not. The Data Lake is closely integrated with data-intensive computation frameworks like Apache Hadoop [**16**], Apache Spark [**108**] and Apache Storm [**99**] to perform data analysis. The vision for the Data Lake is data from numerous sources dropped into the Lake quickly and easily, with tools "fishing" along the edges of the lake, intent on catching insight by the rich ecosystem of data within the lake.

Even though this higher flexibility in Data Lake leads to rich collections of data from various sources, it leaves greater manageability burdens on the hands of scientists. The Data Lake almost becomes a "dump everything" place due to lack of enforced schema. A data item in a Data Lake can be in different stages in its life cycle. It may be in raw stage just after coming out of a data source or may be the result of analysis by one or more of the "fishermen" analysis tools. Frequently products in the Data Lake are the results from one transformation intended for another transformation for further processing. This complicated life cycle of a data product in a Data Lake increases the requirement of proper traceability mechanisms. Without some level of organization, the Data Lake will turn into a data swamp [**30**]. We concentrate on mechanisms which lead to better management of Data Lakes to keep instances from turning into swamps. Critical focus of our attention is

on metadata and lineage information through a data life cycle which are key to good data accessibility and traceability [**56**].

In order to avoid data swamps, Data Lakes should answer questions like "how was this data item derived?, from where did it come from?", "what transformations applied on this data item?, what other data items generated from it?". Provenance can help Data Lakes to achieve that. If a Data Lake can ensure that every data product stored is connected with its provenance information starting from the origin, critical traceability can be had. However, ensuring that we capture all provenance information starting from the origin of a data product in a Data Lake is challenging because a data product may go through different processing systems during its life cycle inside the Data Lake. Most of the time, DIC frameworks like Hadoop, Spark and Storm do not produce provenance information by default. Even if there are provenance collection techniques for those systems, they may use their own ways of storing provenance or use different standards. Therefore generating integrated provenance traces is tough.

We argue that depending on provenance generated from various DIC frameworks used around a Data Lake and trying to integrate them to come up with end to end provenance traces is not viable due to lack of provenance support in most systems and difficulties in Big Data provenance integration. As a solution, we propose a reference architecture based on a central provenance subsystem in the Data Lake environment which stores and processes provenance events pumped into it from all connected systems. We presented our reference architecture as an early work poster [**92**] and construct the current work based on that. Our prototype implementation of the architecture using distributed provenance collection tools

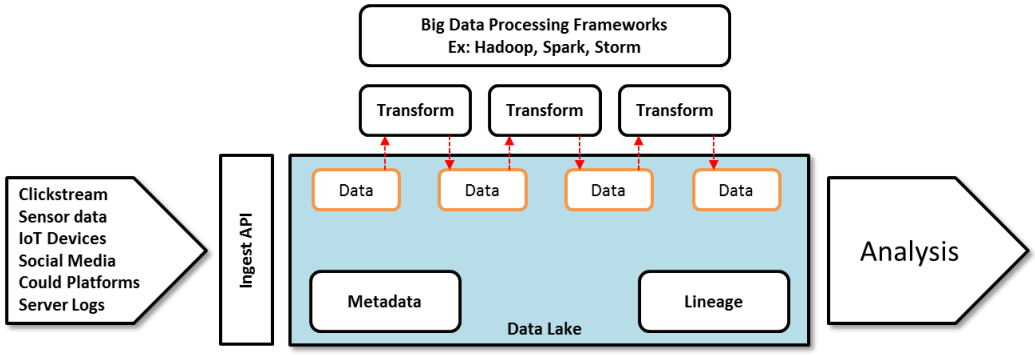


FIGURE 4.4. Data Lake Architecture

shows that the proposed technique can be introduced into a Data Lake to capture integrated provenance without introducing much overhead.

4.2.1. Data Lake Architecture. In most organizations, it is a common practice to maintain a single Data Lake instance which is shared by all the scientists from different departments. The Data Lake is designed to be schema-on-read so that scientists can ingest various types of data into the lake even without having a clear idea about the type of analysis that will be performed. That avoids the loss of information at ingest time.

The general architecture of a Data Lake, shown in Figure 4.4, contains three main activities: (1) Data ingest, (2) Data processing or transformation and (3) Data analysis. A Data Lake may open up number of ingest APIs to bring data from different sources into the lake. In most cases raw data is ingested into the Data Lake for later usage. Different scientists may use the same raw data for different purposes at different points of time. The most important activity in a Data Lake environment is the data transformation. These are the lake's "fishermen", which take data in the Data Lake as inputs and store the output data back into the Data Lake. Modern large scale distributed Big Data processing frameworks like Hadoop, Spark and Storm are used commonly for such transformations. Mechanisms

like scientific workflow systems (Kepler [**10**], Taverna [**75**]) and legacy scripts may apply as well. As shown in Figure 4.4, a data product which has been created as an output from one transformation can be an input into another transformation and that may produce another one as a result. There is no limit to this processing chain as different data scientists within the organization may perform different experiments using data stored in the Data Lake. Finally when all processing steps are done, resulting data products are used for different kinds of analysis reports and predictions.

Different venders and organizations may use different technologies and tools to build their Data Lakes. Hadoop and HDFS [**85**] based Apache Big Data stack is commonly used for most currently existing Data Lake implementations [**90**]. Scalability of HDFS is used to store large amounts of data and the transformations are mostly run using various tools available in the Apache Big Data stack.

4.2.2. Provenance in Data Lake.

4.2.2.1. *Role of Provenance.* The Data Lake achieves increased flexibility at the cost of reduced manageability. When various contributors (i.e., researchers within a community but working on different projects) ingest differently structured data through different APIs, tracking becomes an issue. In addition, chained transformations continuously derive new data from existing data in the lake. In this situation, the Data Lake has to have a mechanism to figure out information like origins and owners of the data products, rights to and suitability of transformations applied to them, rights, ownership and quality of data generated by the transformations. Without having this deep level of traceability, recovery in the event of failures and data reuse become problematic. Selective data provenance can

solve this problem and increase the traceability and accessibility of data within a Data Lake. Therefore, scientists identify provenance as an essential ingredient [**56**] for a Data Lake. Here we discuss several use cases to motivate the study of provenance in a Data Lake.

Use Case 1: A researcher has ingested a dataset into their community Data Lake and later realizes that it contains sensitive data which should not be shared. They wish to delete the dataset or move it to some other secure storage. However by the time they identify the security risk, there are other scientists who have already used this data in subsequent experiments. Output data schemas of such transformations may have allowed sensitive parts of input data to flow into the results as well. If that is the case, they may need to delete such derived data as well. In this situation, how can the scientists find who used the original dataset, what transformations were performed on them, what results were generated and where are they stored? All these questions can be answered by using integrated provenance across transformations starting from the origins. A forward provenance query on the original data identifier can be used here to get all needed information.

Use Case 2: When analyzing a certain dataset found in their Data Lake, a researcher sees an odd result and wishes to figure out how this dataset was generated. By tracing the input data to the methodology source, which is possibly located outside the Data Lake in a publications repository, the researcher sees that the dataset had been cleaned of data points on the basis that the data points were spurious, when in reality they were important outlier data points. A backward provenance query on their data product gives the researcher grounds for discarding their result and seeking a more fitting input dataset.

Use Case 3: A Data Lake could be used as a staging area [**90**] for observational data before a researcher moves their data into a more structured data store. The researcher could use the Data Lake for purification and aggregation of data prior to writing the data into a more structured format. When exporting a transformed dataset from the Data Lake, the derivation history of activity within the Data Lake will need to be linked with the dataset so that future experiments can use it. Attaching provenance traces generated for exported datasets provides a good solution in this situation.

These use cases delineate the new provenance challenges in the Data Lakes model. This work lays a foundation for answering these questions, though some of the challenges identified remain as future work. In addition to that, data provenance captured at required granularity can be used in other traditional use cases like debugging and reproducing transformations, performance analysis and experiment monitoring and visualization within a Data Lake.

4.2.2.2. *Challenges in Provenance Capture.* Data in a Data Lake may go through number of transformations performed using different DIC frameworks selected according to the type of data and application. For example, in a HDFS based Data Lake, it is common to use Storm or Spark Streaming for streaming data and Hadoop MapReduce or Spark for batch data. Other legacy systems and scripts may be included as well. To achieve traceability across transformations, provenance captured from these systems must be integrated, a challenge since many do not support provenance by default.

Techniques exist to collect provenance from Big Data processing frameworks like Hadoop and Spark [**5**, **54**, **77**, **104**]. But most are coupled to a particular framework. If

the provenance collection within a Data Lake depends on such system specific methods, provenance from all subsystems should be stitched together to create a deeper provenance trace. There are stitching techniques [**66**, **76**] which bring all provenance traces into a common model and then integrate them together. However the process of converting provenance traces from different standards into a common model may lose provenance information depending on the data model followed by each standard. As a Data Lake deals with Big Data, most transformations generate large provenance graphs. Converting such large provenance graphs into a common model and stitching them together can introduce considerable compute overheads as well.

4.2.2.3. *Provenance Integration Across Systems.* To address provenance integration, we propose a central provenance collection system to which all components within the Data Lake stream provenance events. Well accepted provenance standards like W3C PROV [**70**] and OPM [**69**] represent provenance as a directed acyclic graph ($G = (V, E)$). A node ($v \in V$) can be an activity, entity or agent while an edge ($e = \langle v_i, v_j \rangle$ where $e \in E$ and $v_i, v_j \in V$) represents a relationship between two nodes. In our provenance collection model, a provenance event always represents an edge in the provenance graph. For example, if process p generates the data product d, the provenance event adds a new edge ($e = \langle p, d \rangle$ where $p, d \in V$) into the provenance graph to represent the 'generation' relationship between activity p and entity d.

In addition to capturing usage and generation, additional details like configuration parameters and environment information (*e.g.*, CPU speed, memory capacity, network

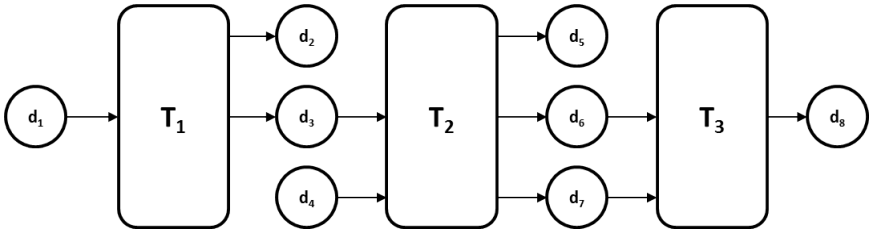


FIGURE 4.5. A high level data flow scenario in a Data Lake

bandwidth) can be stored as attributes connected to the transformation. Inside each transformation, there can be number of intermediate tasks which may themselves generate intermediate data products. A MapReduce job for instance has multiple map and reduce tasks. Capturing provenance from such internal tasks at a high enough level to be useful helps in debugging and reproducing transformations.

When the output data from one analysis tool is used as the input to another, integration of provenance collected from both transformations can be guaranteed only by a consistent lake-unique persistent ID policy [**70**]. This may require a global policy enforced for all contributing organizations to a Data Lake. This unique ID notion could be based on file URLs or randomly generated data identifiers which are appended to data records when producing outputs so that the following transformations can use the same identifiers. It could also be achieved using globally persistent IDs such as the Handle system or DOIs. As a simple example, consider Figure 4.5. The data product d_1 is subject to transformation T_1 and generates d_2 and d_3 as results. T_2 uses d_3 together with a new data product d_4 and generates d_5, d_6 and d_7. Finally T_3 uses d_6 and d_7 and generates d_8 as the final output. When all three transformations T_1, T_2 and T_3 have sent provenance events, a complete provenance graph is created in the central provenance collection system. Figure 4.6 shows the high level data lineage graph which represents the data flow starting from d_8.

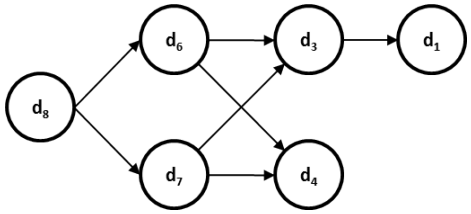


FIGURE 4.6. Data Lineage

4.2.2.4. *Reference Architecture.* The reference architecture, shown in Figure 1.1 in Chapter 1, uses a central provenance collection subsystem. Provenance events captured from components in the Data Lake are streamed into the provenance subsystem where they are processed, stored and analysed. The Provenance Stream Processing and Storage component at the heart of this architecture accepts the stream of provenance notifications (Ingest API) and supports queries (Query API). A live stream processing subsystem supports live queries while storage subsystem persists provenance for long term usage. When long running experiments in the Data Lake produce large volumes of provenance data, stream processing techniques become extremely useful as storing full provenance is not feasible. The Messaging System guarantees reliable provenance event delivery into the central provenance processing layer. Usage subsystem shows how provenance collected around the Data Lake can be used for different purposes. Both live and post-execution queries over collected provenance with Monitoring and Visualization helps in scenarios like the use cases that we discussed above. There are other advantages as well such as Debugging and Reproducing experiments in the Data Lake.

In order to capture information about the origins of data, provenance must be captured at the Ingest. Some data products may carry their previous provenance information which should be integrated as well. Researchers may export data products from the Data Lake in

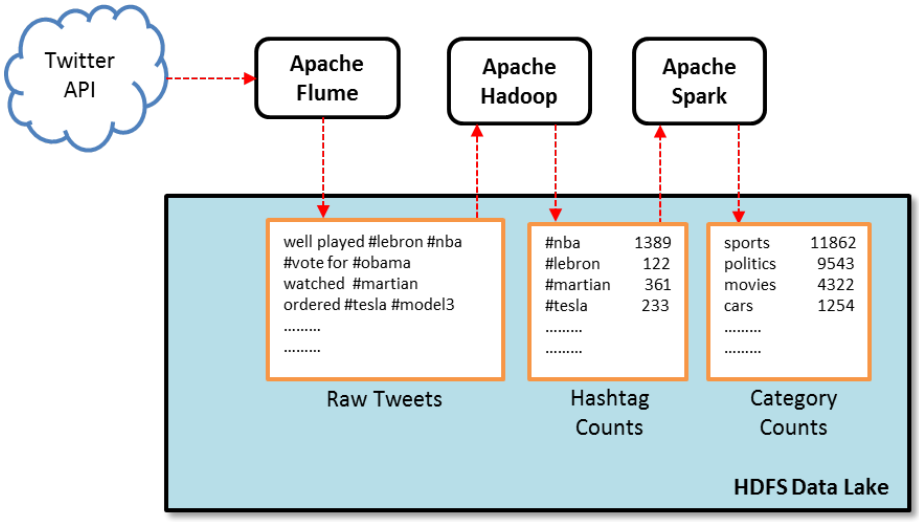


FIGURE 4.7. Data Lake use case

some situations. Such data products should be coupled with their provenance for better usage.

4.2.3. Prototype Implementation. We set up a prototype Data Lake and implemented a use case on top of it to evaluate the feasibility of our reference architecture. We used our provenance collection tools to capture, store, query and visualize provenance in our Data Lake. The reference architecture introduces both stored provenance processing and real time provenance processing for Data Lakes. In this prototype, we implement stored provenance processing; real time provenance processing is addressed as the main topic of this thesis. The central provenance subsystem uses our Komadu [**93**] provenance collection framework.

4.2.3.1. *Data Lake Use Case.* The Data Lake prototype was implemented using an HDFS cluster and the transformations were performed using Hadoop and Spark. Analysing data from social media to identify trends is commonly seen in Data Lakes. As shown in Figure

4.7, we have implemented a chain of transformations based on Twitter data to first count hash tags and then to get aggregated counts based on categories. Apache Flume **[15]** was used to collect Twitter data and store in HDFS through the Twitter public streaming API. For each tweet, Flume captures the Twitter handle of the author, time, language and the full message and writes a record into an HDFS file. After collecting Twitter data over a period of five days, a Hadoop job was used to count hash tags in the full Twitter dataset. A new HDFS file with hash tag counts is generated as the result of the first Hadoop job which is used by a separate Spark job to get aggregated counts according to categories (sports, movies, politics etc). We just used a fixed set of categories for this prototype implementation to make it simple. In real Data Lakes, these transformations can be performed by different scientists at different times. They may use DIC frameworks based on their preference and expertise. That is why we used two different frameworks for the transformations in our prototype to show how provenance can be integrated across systems.

Komadu and its tool kit was used to build the provenance subsystem (shown in Figure 1.1) in our prototype. Komadu supports RabbitMQ messaging system and includes tools to fetch provenance notifications from RabbitMQ queues. A RabbitMQ instance was deployed in front of our Komadu instance so that all provenance notifications generated by Flume, Hadoop and Spark goes through a message queue in RabbitMQ. Ingested provenance events are asynchronously processed by Komadu and stored in relational tables. Stored provenance remains as a collection of edges until a graph generation request comes in. This delayed graph generation leads to efficient provenance ingest with minimum back pressure. This helps in a Data Lake environment where high volumes of provenance are generated.

To assign consistent identifiers for data items in our Data Lake, we followed the practice of appending identifies to data records when output data is written to the Data Lake. Subsequent transformation uses the same identifiers for provenance collection. Provenance events were captured in our prototype by instrumenting the application code that we implemented for each transformation. Tweet capturing code in Flume was instrumented to capture provenance at the data ingest into the Data Lake. *Map* and *Reduce* functions in the Hadoop job and *MapToPair* and *ReduceByKey* functions in the Spark job were instrumented to capture provenance from transformations. We implemented a client library with a simple API (like Log4J API for logging) which can be used to easily instrument Java applications for provenance capture. It minimizes the provenance capturing overhead by using a dedicated thread pool to asynchronously send provenance events into the provenance subsystem. In addition to that, the client library uses an event batching mechanism to minimize the network overhead by reducing the number of messages sent into the provenance subsystem over the network.

4.2.3.2. *Provenance Queries and Visualization.* After executing the provenance enabled Hadoop and Spark jobs on collected Twitter data, Komadu query API was used to generate provenance graphs. Komadu generates PROV-XML provenance graphs and it comes with a Cytoscape plugin which can be used to visualize and explore them. Fine-grained provenance includes input and output datasets for each transformation, intermediate function executions and all intermediate data products generated during the execution. Provenance from Flume, Hadoop and Spark have been integrated together through the usage of unique data identifiers.

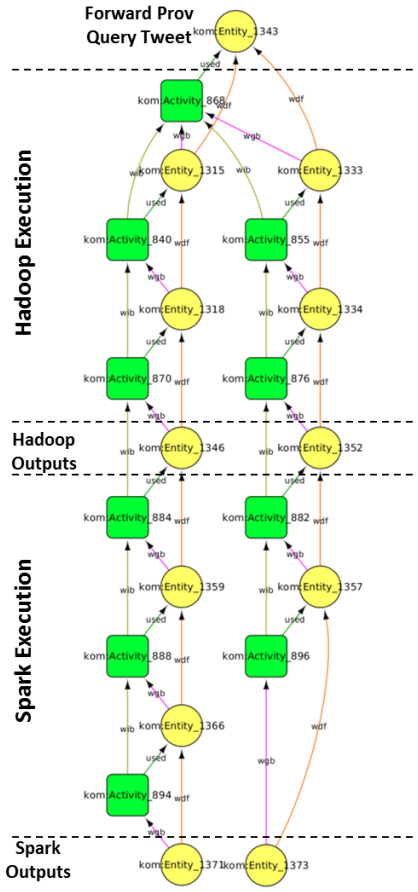


FIGURE 4.8. Cytoscape visualization of forward provenance for a single tweet

Forward provenance is useful to derive details about the usages of a particular data item. Figure 4.8 shows a forward provenance graph for a single tweet. It shows the hash tags generated by that particular tweet in Hadoop outputs and the categories to which those hash tags contributed in Spark outputs. A backward provenance graph starting from a category under Spark outputs is shown in Figure 4.9 (figures only show a small subset of data to visualize clearly). This graph can be used to find all tweets which contributed for that category. For example, if a scientist wanted to get an age distribution of the authors

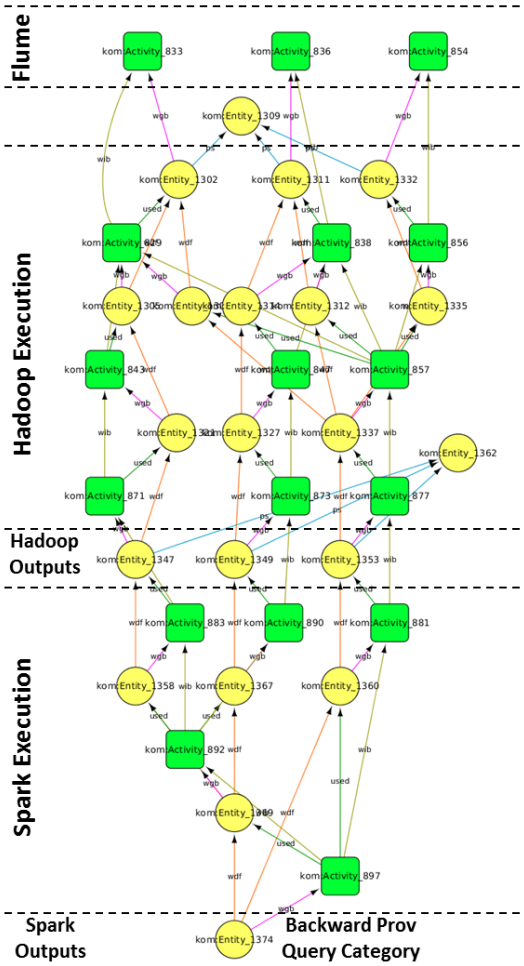


FIGURE 4.9. Cytoscape visualization of backward Provenance for one Spark output

who tweeted about sports, it can be done by finding the set of Twitter handles of the authors through backward provenance.

4.2.3.3. *Performance Evaluation.* To build our prototype, we used five small VM instances with 2 CPU cores of 2.5GHz speed, 4 GB of RAM and 50 GB local storage on each instance. Four instances were used for the HDFS cluster including one master node and three slave nodes. Total of 3.23 GB Twitter data was collected over a period of five days by running

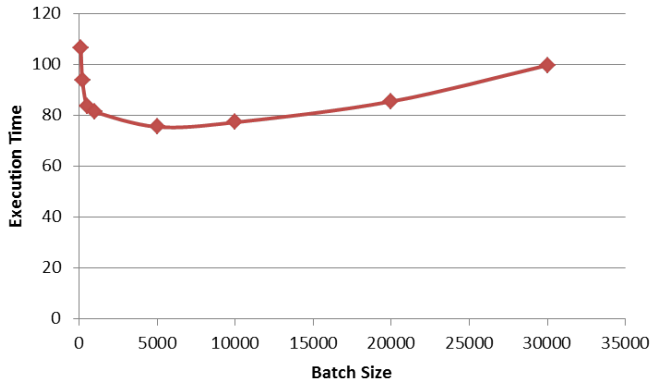


FIGURE 4.10. Execution time with varying batch size

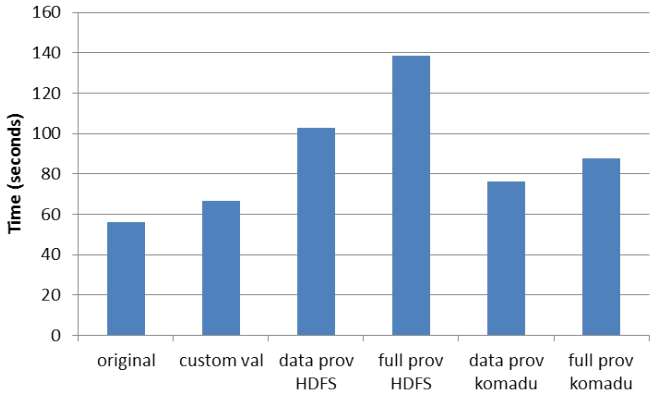


FIGURE 4.11. Hadoop Execution times for different scenarios

Flume on the master node. Hadoop and Spark clusters were set up on top of our four node HDFS cluster. One separate instance was allocated to set up the provenance subsystem using RabbitMQ and Komadu tools.

In order to minimize the provenance capture overhead, we used a dedicated thread pool and a provenance event batching mechanism in our client library. When the batch size is set to a relatively large number (>500), execution time becomes almost independent of the thread pool size as the number of messages sent through the network reduces. Therefore, we set the client thread pool size to 5 in each of our experiments. Figure 4.10 shows how

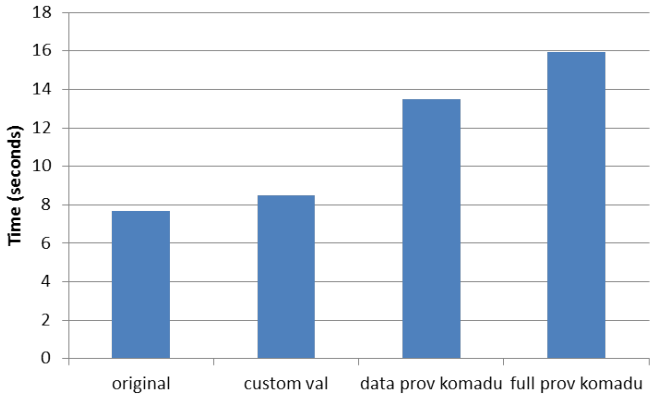


FIGURE 4.12. Spark Execution times for different scenarios

the provenance enabled Hadoop execution time for a particular job varies when the batch size is increased from 100 to 30000 (provenance events). As per this result, we set the batch size to 5000 in each of our experiments. We used JSON format to encode provenance events and the average event size is around 120 bytes. The average size of a batched message sent over the network is around 600 KB (5000 x 120 bytes).

Figure 4.11 shows the execution times of the Hadoop job for different scenarios. Column 'original' represents the Hadoop execution time without capturing any provenance. In order to relate *Map* and *Reduce* provenance, we had to use a customized value field in the output key-value pair from the *Map* function which contains data identifiers. This is similar to the technique used in Ramp [77]. As shown by 'custom val' column in the chart, usage of customized value introduces an overhead of 19.28% and that is included in all other cases. Execution overhead depends on the granularity of provenance as well. Columns 'data prov komadu' and 'full prov komadu' shows the execution times of Hadoop when our technique is used to capture provenance. Data provenance (data relationships only) case adds a 36.47% overhead while full (data and process relationships) provenance case adds a

TABLE 4.1. Size (in GB) of provenance generated by Hadoop

	Map	Combine	Reduce	Total
Data Provenance	3.232	1.281	0.529	5.042
Full Provenance	9.733	1.824	0.813	12.37

56.93% overhead. Table 4.1 shows a breakdown of provenance sizes generated for each case in Hadoop for the input size of 3.23 GB. Size of provenance doubles for full provenance case compared to data provenance and that leads to greater capturing overheads. As it is a common practice [**77**] to write provenance into HDFS in Hadoop jobs, we modified the same Hadoop job to store provenance events in HDFS as well and compared the overhead with our method. As shown by 'data prov HDFS' and 'full prov HDFS' columns in Figure 4.11, that adds larger overheads compared to our techniques. Better performance have been achieved by modifying or extending Hadoop [**5**]. But our techniques operate completely on application level without modifying existing DIC frameworks.

Figure 4.12 shows the execution times for the Spark job for different scenarios. Like in Hadoop, we used a customized output value to include data identifiers in Spark as well. That adds an overhead of 7.5% compared to original execution time as shown by 'custom val' column. Data provenance and full provenance cases using Komadu add overheads of 76.1% and 108.35% respectively. Overhead percentages added by provenance capture in Spark is larger compared to Hadoop as Spark works faster than Hadoop and our techniques introduce same level of overhead in both cases.

4.2.4. Conclusion. In this work first we highlighted the importance of data provenance for a Data Lake by presenting specific use cases. Then we discussed the challenges in applying provenance in a Data Lake and came up with a reference architecture to overcome

those challenges. Then we presented a prototype implementation of our architecture and showed how integrated provenance across DIC frameworks and tools can be achieved. Finally we discussed the techniques we used to minimize the instrumentation overhead and presented a performance evaluation.

We implemented only the stored provenance processing techniques in the presented prototype. Our reference architecture highlights the power of real-time provenance processing in Data Lakes and we address it as the main topic in this thesis. As seen in our results, fine-grained provenance can be multiple times larger than the input datasets in Data Lakes and handling such "Big Provenance" is a challenge. Storing and querying such volumes of provenance is not feasible and that motivates real-time provenance stream processing.

CHAPTER 5

Provenance Stream Model

This chapter provides the theoretical foundation of the thesis. As our provenance stream processing techniques are focused on provenance generated from Data-Intensive Computations (DIC), first we formally define DICs, DIC workflows and backward and forward provenance for DICs. Our definitions are built on specific properties of DICs which are also discussed here. A general definition for a stream of provenance also presented to complete our provenance stream model.

5.1. DICs and DIC Workflows

As described in Chapter 3.2, there are two main categories of Data-Intensive Computations: Batch computations and Stream computations. A batch computation applies a set of functions on a stored static data set and terminates in finite time. A stream computation applies a set of functions on a stream of data elements and executes for a longer period of time. Both batch processing and stream processing DICs can be described as a directed acyclic graph (DAG) of functions that executes on a framework such as Hadoop or Spark. Each function consumes a set of data products as inputs and produces another set of data products as outputs. An output from one function is fed into the next function in the DAG as an input.

Definition: *A data-intensive computation (DIC) with input D^i is a directed acyclic graph of n functions F_1, F_2, .., F_n such that $F_k(D^i_k) = D^o_k; 1 \leq k \leq n$ where $D^i_k \subseteq \{D^i \cup D^*\}$, D^* is the union of outputs from already executed functions in the DIC and the final output, $D^o \subseteq \cup^n_{k=1} D^o_k$.*

Each function F_k is executed in parallel on partitions of its input data ($D^i_k = d^i_{k1} \cup d^i_{k2} ... \cup d^i_{kp}$) satisfying the following condition. We call such an execution $f_{kj}(d^i_{kj})$ a *function execution*.

$$F_k(D^i_k) = f_{k1}(d^i_{k1}) \cup f_{k2}(d^i_{k2}) ... \cup f_{kp}(d^i_{kp})$$

Each function execution consumes a set of input data products and produces another set of output data products. As a DIC is a DAG of functions, there are intermediate data products generated between internal function executions. Therefore, data products can be categorized as input data products, intermediate data products and output data products. From a provenance perspective, it is important to define a path between an input data product and an output data product.

Definition: *A path between an input data product and an output data product in a DIC is a sequence of n ordered function executions f_1, f_2, ..., f_n which satisfy the following two conditions. d^i_k is the input data product and d^o_k is the output data product for function execution f_k. d^i_1 is the input data product and d^o_n is the output data product between which the path is considered.*

$$f_k(d^i_k) = d^o_k; 1 \leq k \leq n$$

$$d^i_k = d^o_{k-1}; 1 < k \leq n$$

MapReduce model **[36]** supports functions: map, combine, and reduce. During early days of Hadoop MapReduce, each computation was just a sequence of functions where

outputs from one function are fed into the next function. However, with the additions like multiple input support, Hadoop is able to execute DAGs of functions. Modern DIC frameworks like Spark and Flink have a broader set of in-built functions: map, filter, reduce, sortByKey, join etc. which supports complex DAGs of functions both on batch mode and streaming mode. These frameworks assign multiple worker nodes to execute a function in parallel on different partitions of input data.

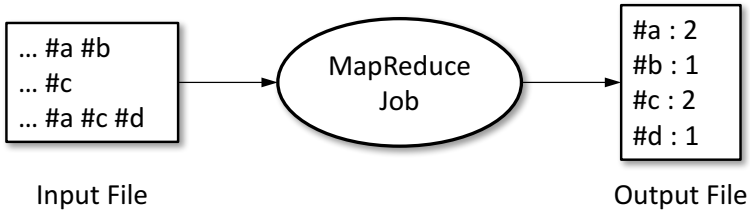


FIGURE 5.1. MapReduce DIC for hashtag counting

Figure 5.1 shows an example of a MapReduce DIC that counts hashtags in a set of tweets from an input file. When the job is executed, each line in the input file is fed into a map function which outputs the key-value pair $\langle hashtag, 1 \rangle$ for each hashtag found in that line. Once all map functions are completed, all key-value pairs emitted by map functions are shuffled and combined by their keys and fed into the reduce function. It calculates the total number of occurrences for each key (hashtag in this example) and produces the output file shown.

A DIC workflow may consist of one or more DICs depending on the number of steps in the workflow. Often multiple steps like data importing, data cleansing, joining, transforming, filtering, exporting etc. are included in a DIC workflow. Each processing step in a DIC workflow is performed using a DIC framework such as Hadoop, Spark or Flink.

There are DIC workflow execution frameworks such as Apache Oozie [**19**] and Apache Nifi [**18**]. Oozie provides a high level XML based language which can be used to define a DIC workflow which consists of DICs running on multiple frameworks from the Apache big data stack. Apache Pig [**20**] and Apache Hive [**17**] also can be seen as workflow tools where Pig provides a scripting language to define Hadoop workflows and Hive supports a SQL syntax to define Hadoop workflows. Irrespective of the workflow tool used, data processing steps are always executed by a DIC framework.

Definition: *A DIC workflow with input* D^i *is a set of m DICs* $C_1, C_2, .., C_m$ *such that* $C_k(D^i_k) = D^o_k; 1 \leq k \leq m$ *where* $D^i_k \subseteq \{D^i \cup D^*\}$, D^* *is the union of outputs from already completed DICs in the workflow and the final output,* $D^o \subseteq \cup_{k=1}^{m} D^o_k$.

In simple terms, a DIC workflow is a DAG of DICs. Some DICs within a workflow may execute in parallel and they may consume data from the inputs to the workflow and outputs from the other DICs. The final output of the workflow may consist of the outputs from one or more DICs. The input data to a DIC workflow often consists of differently typed data from different sources. Figure 5.2 shows a DIC workflow which consists of six DICs operating on four partitions of input data.

5.2. Backward and Forward Provenance

Fine-grained provenance from a DIC consists of input and output data products of all internal functions and their relationships. Provenance graph for above MapReduce example is shown in Figure 5.3. The graph mainly shows the W3C Prov derivation relationships, *used* and *wasGeneratedBy*. For example, the map function execution M_1 uses the data product i_1 (".. #a #b") and generates data products n_1 (#a:1) and n_2 (#b:1).

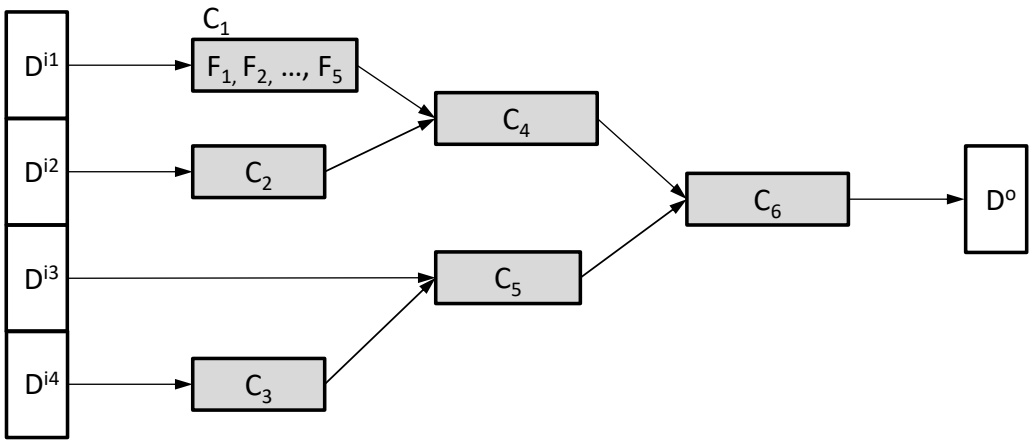


FIGURE 5.2. DIC workflow made up of six DICs

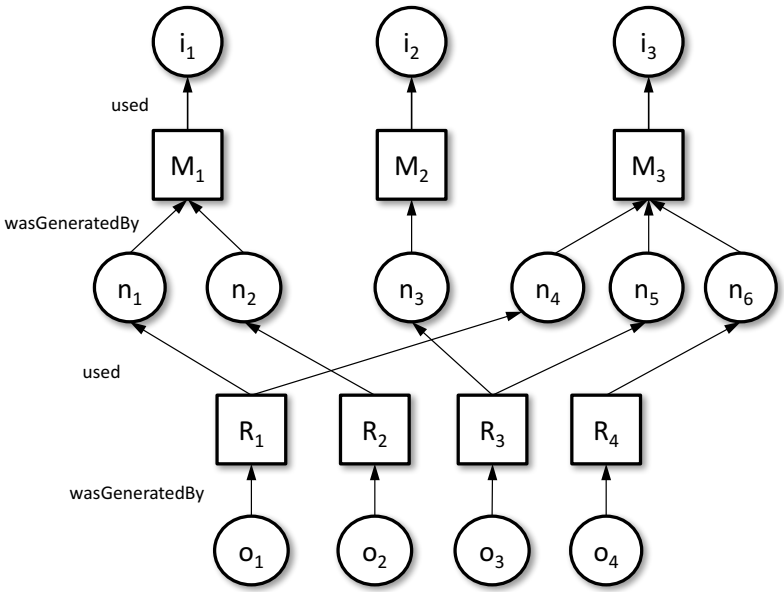


FIGURE 5.3. Provenance Graph for MapReduce job in 5.1 with Edge Types and Identifiers

Backward provenance and forward provenance define a minimal but sufficient type of provenance needed for several useful provenance analysis tasks **[29]** **[28]**. Backward provenance begins with an output and traces backwards in time to the subset of input data on which the output depends. Forward provenance begins with an input and traces

forward in time to the subset of output data elements derived by it. Whereas RAMP [77] [53] recursively defines provenance for any data element in MapReduce, we generalize the definition for any data element in a DIC.

Definition: *One-function backward provenance of output element o from function f_i of a DIC is the set E of intermediate elements contributing as input to f_i. Backward provenance of o is then the recursive union of the backward provenance for each $e \in E$. Recursivity terminates when all inputs e are elements in input data for the DIC.*

Definition: *One-function forward provenance of input element i to function f_j of a DIC is the set E of intermediate elements produced as output by f_j. Forward provenance of i is then the recursive union of the forward provenance for each $e \in E$. Recursivity terminates when all outputs e are elements in output data from the DIC.*

Backward provenance then for an output or intermediate data element in a DIC is the subset of input data elements (to the DIC) on which it depends. Forward provenance for an input or intermediate data element in a DIC is the subset of output data elements (from the DIC) derived by it. The intermediate data is between functions in the context of a DIC.

Having established the data-intensive computation (DIC) as the basic building block of a DIC workflow, we can extend the above definitions to reason about backward and forward provenance for the DIC workflow as a whole. In a workflow, a dependency path between an input data element and an output data element may go through one or more DICs as illustrated in Figure 5.2. Backward provenance then for an output or intermediate data element in a DIC workflow is the subset of input data elements (to the workflow) on which it depends. Forward provenance for an input or intermediate data element in a DIC

workflow is the subset of output data elements (from the workflow) derived by it. The intermediate data is between DICs in the context of a DIC workflow.

5.3. Provenance Streams

The two widely used provenance representation languages, OPM [**69**] and W3C PROV [**67**] which we use, both represent provenance as a directed acyclic graph. Provenance generated by each DIC in a DIC workflow corresponds to a single provenance graph.

Definition: *A Provenance Graph G = (V, E, A) is a directed, acyclic graph where a node ($v \in V$) is an activity, entity, or agent defined in W3C PROV, an edge ($e = \langle v_i, v_j \rangle$ where $e \in E$ and v_i, $v_j \in V$) represents a relationship defined in W3C PROV directed from v_i to v_j and a set of attributes $A(p) = \{a_1, a_2, ...\}$ belongs to node or edge p.*

A provenance stream can be thought of as a serialization of a static provenance graph. A provenance stream for a DIC is created on-the-fly during execution of a DIC. Elements that grow a provenance graph on-the-fly correspond to provenance relationships (edges) being established (e.g., use of a particular data product). Frequently a node's existence is asserted upon its first use.

Definition: *A Provenance Stream $S = \{s_1, s_2, ..., s_n\}$ representing a Provenance Graph G = (V, E, A) is an append-only sequence of elements where an element s represents one of the following.*

(1) $s \in E$

(2) $s = \langle P_m \rangle$ *where* $m \in V$ *or* $m \in E$, *m is already found in the stream before s and* $P_m \subseteq A(m)$

An element in a provenance stream is a provenance relationship asserted between two vertices or a set of attributes for either a vertex or an edge that has already appeared in the

stream before the current element. Attributes are allowed for the edge elements when they are initially created. However, in some situations, attributes need to be added later. For example, when a function execution starts, the start time can be recorded as an attribute in the very first edge which uses the function. However, the end time can only be added after the function execution has completed. Consider the example shown in Figure 5.4 in which the end time is captured as a new attribute later in the stream. Another approach to capture start time and end time for an activity is to use *wasStartedBy* and *wasEndedBy* relationships defined in W3C PROV.

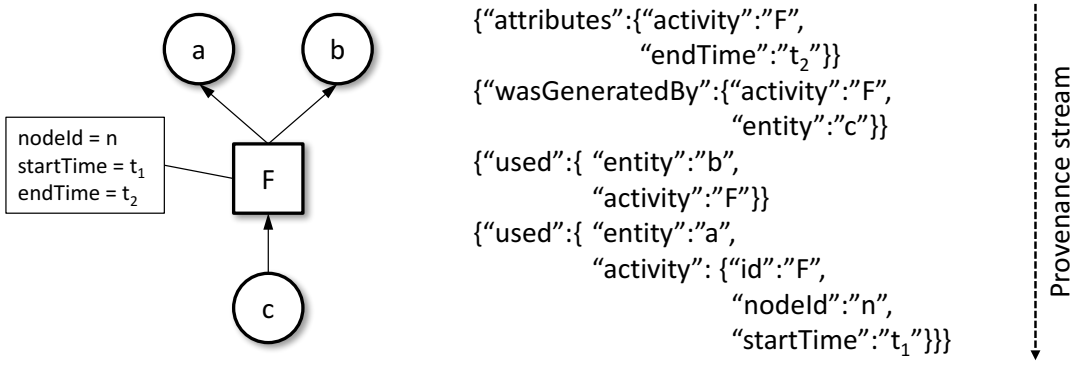


FIGURE 5.4. Adding new Attributes to an Existing Node

The provenance stream model defined in [**28**] allows only derivation relationships that are temporally ordered. Here we extend their definition to allow other relationships and accommodate events out-of-order.

Each DIC in a DIC workflow generates a separate provenance graph stream. We call a stream of raw provenance (before processing) a *full provenance graph stream*. We apply our parallel provenance stream processing algorithm on a full provenance graph stream generated by a single DIC to reduce the amount of provenance on-the-fly while

preserving backward and forward provenance. Each reduced provenance stream is stored in a provenance repository for archiving and querying.

CHAPTER 6

Parallel Provenance Stream Processing

Having defined backward provenance and forward provenance for DICs and DIC workflows in the previous chapter, here first we discuss the "Big Provenance" problem in the context of large-scale DICs. Then we elaborate on how on-the-fly provenance analysis helps in mitigating the problems caused by Big Provenance. Then we move onto our parallel provenance stream processing algorithm and its application in a provenance stream processing architecture. Several provenance stream partitioning strategies are also presented to wrap up the chapter.

6.1. Big Provenance in DICs

Fine-grained provenance captured from DICs is useful for debugging and monitoring computations, for tracing the origins of derived data, and for tracing the derivation paths for input data. In order to analyze how each input was processed and how each output was generated, details on each function execution should be recorded including their input and output data products. Fine-grained provenance collected from both map and reduce functions can fulfill that requirement.

Figure 5.1 from Chapter 5 shows a simple example of a MapReduce DIC that counts hashtags in a set of tweets from an input file. When the job is executed, each line in the input file is fed into a map function which outputs $\langle hashtag, 1 \rangle$ for each hashtag found in

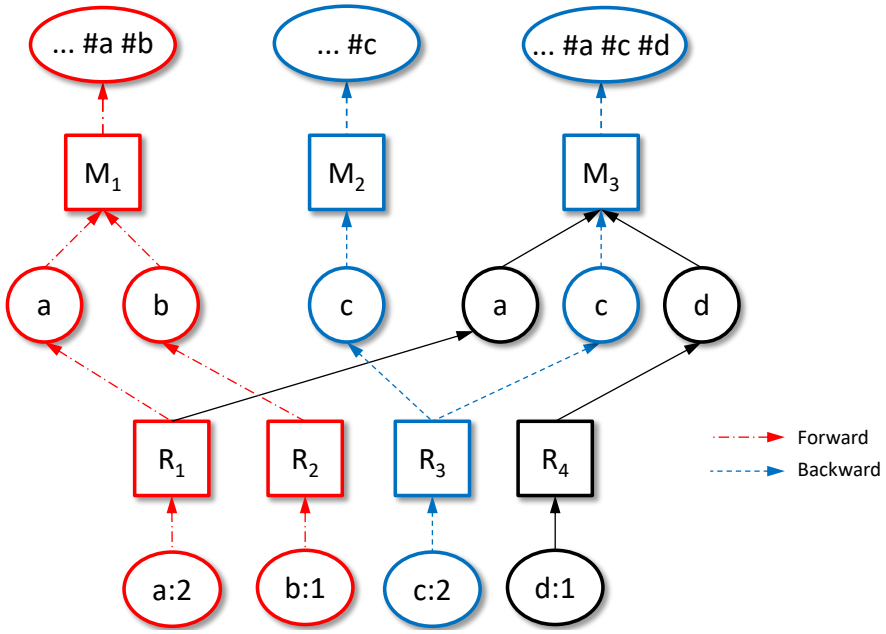


FIGURE 6.1. Full provenance for MapReduce example illustrating forward provenance for first tweet and backward provenance for output $c:2$. Direction of arrows follow W3C Prov convention for usage and generation.

that line. Reduce function calculates the total number of occurrences for each hashtag and produces the output file shown in the figure. Figure 6.1 shows the full provenance graph highlighting backward and forward provenance for the above example including inputs and outputs for all function executions. Backward provenance (shown in blue) for output $c:2$ is derived by recursively traversing the graph from leaf node until the root nodes are found. Forward provenance (shown in red) for the first tweet (input to M_1) is derived by recursively traversing the graph from the root node until the leaf nodes are found.

$$backward - provenance(c:2) = \{''\ldots\ \#c'',\ ''\ldots\ \#a\ \#c\ \#d''\}$$

$$forward - provenance(''\ldots\ \#a\ \#b'') = \{a:2,\ b:1\}$$

As we defined above, backward and forward provenance for a DIC shows dependencies between the inputs to the first function and the outputs from the last function. It is important to note that provenance of intermediate functions is used only to derive paths between inputs and outputs of the DIC when calculating backward and forward provenance. For a large-scale computation which uses multiple functions, storing provenance from such intermediate functions contributes heavily towards the storage and query issues. Our focus is to remove intermediate provenance on-the-fly before provenance is stored in a repository.

Backward and forward provenance is useful in many different scenarios. For an example, if an increase in interest for a certain product from a manufacturer is seen as a result of a twitter data analysis workflow, backward provenance can locate the subset of input tweets which contributed to the result for further analysis like users' geographic distributions, age distributions etc. Forward provenance is useful in cases like tracing all output records which were derived by some corrupted records in the input. This tracing is important to clean-up the results derived by corrupted input data.

Here we focus only on provenance analysis based on backward and forward provenance and get rid of intermediate provenance while preserving them. Our techniques permanently remove intermediate provenance from the provenance stream and it is not recoverable. However, such intermediate provenance is important for other types of provenance analysis purposes like debugging and failure tracing. Our techniques do not help in such analysis where full provenance must be available.

Earlier efforts to capture and analyze provenance from DICs include RAMP [**53**], which uses a wrapper-based approach to extend Hadoop to capture provenance. As shown in

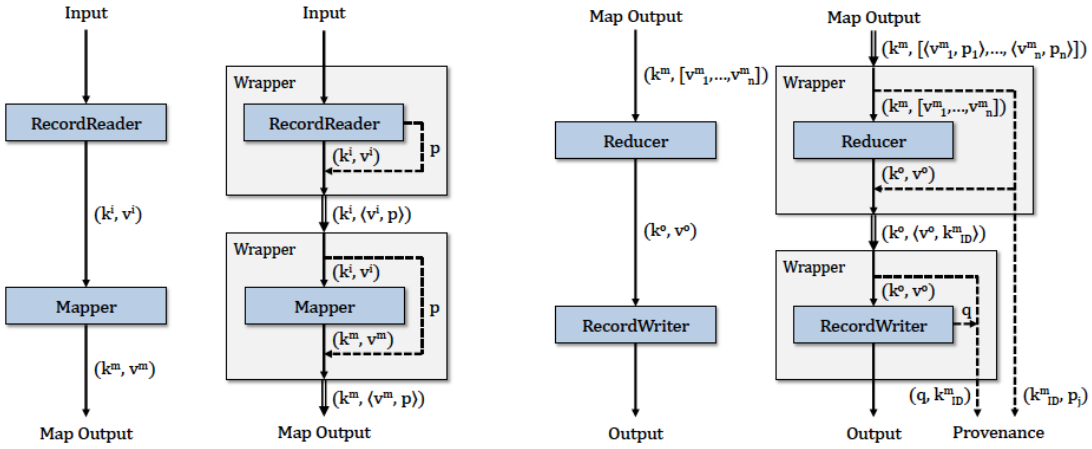


FIGURE 6.2. How RAMP wraps Hadoop to capture provenance. Taken from [77]

Figure 6.2 RAMP propagates input and intermediate data identifiers through a computation and writes provenance into a separate file in HDFS. HadoopProv [**5**] modifies Hadoop instead of extending it and does so to reduce the run-time overhead of capturing provenance. Both solutions persist full provenance information into a storage system. When fine-grained provenance is collected from a DIC workflow, the amount of provenance generated can grow to amounts that challenge most storage solutions. Going beyond relational databases, few recent efforts tackle the volume in provenance stores through techniques utilizing distributed file systems [**110**], NoSQL stores [**1**], and graph databases [**45**].

Irrespective of the techniques used, having to expand the size of the storage layer by multiple times just to store provenance is not realistic and efficient in most applications. Provenance queries frequently require extensive graph traversal: calculating backward/-forward provenance, finding all paths through a given function, and checking whether a given output is dependent on a given input. Graph traversal queries are frequently

supported using recursion, however, recursion is considered extremely slow and compute-intensive [**50**] [**61**] for large graphs. Storing transitive closure tables for each node in a graph is another technique for faster graph traversal. But transitive closure tables consume significant space [**61**] and are computationally expensive.

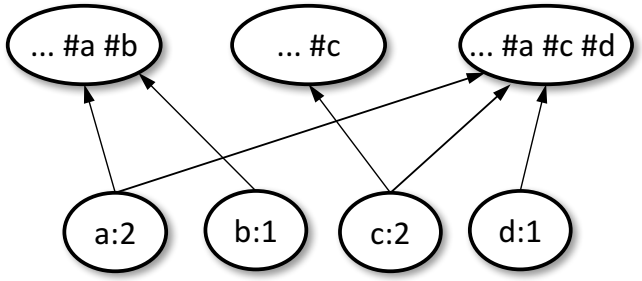


FIGURE 6.3. Reduced backward and forward provenance

6.2. Streaming solution for Big Provenance

In this thesis, we propose a stream processing approach that runs in near real-time to the application to reduce the volume of provenance from a DIC, saving unnecessary writes to storage and reducing query overheads. Our stream processing techniques are applied to a stream of full provenance from a DIC to derive a reduced provenance graph which preserves backward and forward dependency relationships between the inputs and outputs. Intuitively, provenance related to intermediate data products and function executions are removed in real-time and they are only used to maintain dependency paths between inputs and outputs. Figure 6.3 shows the reduced provenance graph which only contains backward and forward provenance for the example in Figure 5.1. For large scale DICs with multiple functions, this reduction in graph size helps with both storage and query efficiency.

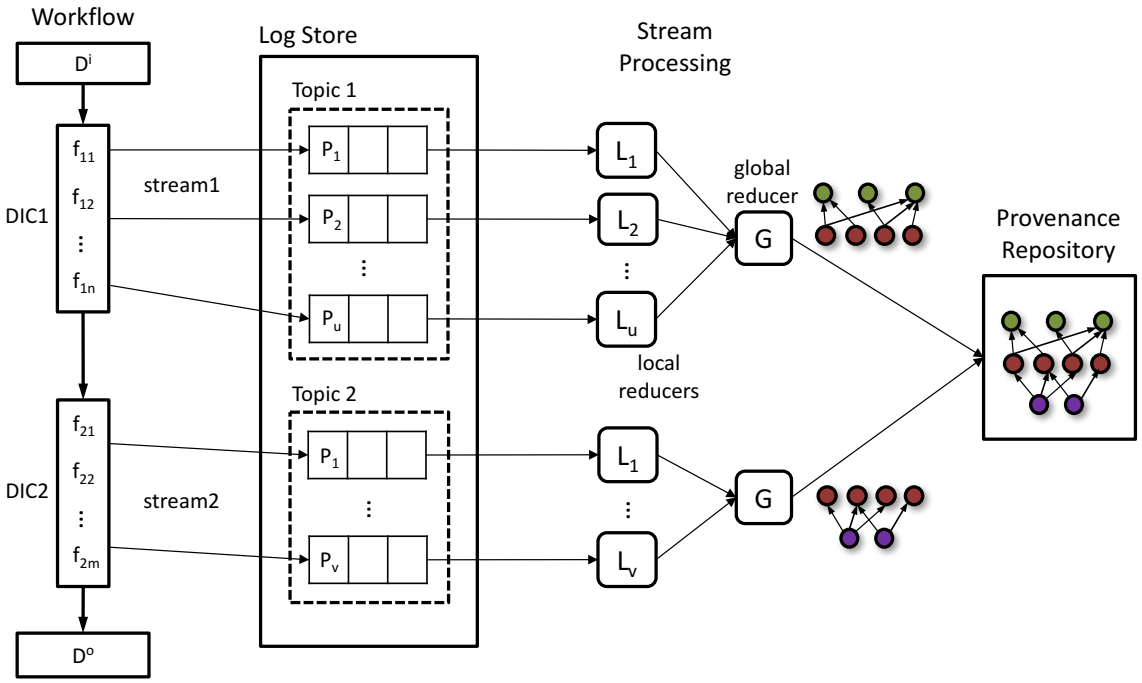


FIGURE 6.4. Stream processing solution for a DIC workflow

Figure 6.4 shows the provenance stream processing solution for a DIC workflow which consists of multiple DICs. Each DIC executes multiple functions which emit provenance elements towards a provenance stream. *DIC1* consumes the input dataset of the workflow, D^i and *DIC2* consumes the output of *DIC1* and produces the final output data set of the workflow, D^o. This diagram shows a simple DIC workflow which only consists of two DICs for the ease of presentation. However, our techniques are applicable for any DIC workflow which follows the definition given in Chapter 5.

Provenance from each DIC in the workflow is considered as a separate stream. Each provenance stream is stored in a "Topic" of a distributed log storage system [**58**] for reliability and ease of consumption. A log store guarantees retention of stream elements for a configurable period of time until they are read by the consumers. Topics support

partitioning so that multiple stream consumers can consume multiple partitions of the same stream in parallel.

Each provenance stream is split into multiple partitions to achieve scalability through parallel stream processing. Each partition is processed by a local stream reducer which uses stateful stream processing. Our *parallel-prov-stream* algorithm (described below in Section 6.3) maintains a reduced set of provenance graph edges as the local state and when a new edge comes in, it is used to perform further reductions in the local state. After processing a configured number of stream elements (*local batch size*), local reducer periodically flushes the reduced local state towards the global reducer.

Global reducer also uses the same algorithm and periodically flushes the reduced state to the provenance repository for permanent storage. The degree of reduction can be increased by increasing the *global batch size* which is the number of provenance edges processed before each flush of the global state. Provenance streams generated by streaming DICs are often infinite or runs for a long time. In that case, *global batch size* should be tuned to maximize the throughput of the system while enabling enough reductions. If the provenance stream is finite and the global reducer has enough resources allocated, *global batch size* can be configured to flush the global state only at the end of the stream. This provides maximum reduction and does not leave any intermediate provenance edges in the final graph.

Reduced provenance graphs from all DICs in a workflow are stored and merged in a central provenance repository to build backward and forward provenance for the entire workflow. Provenance queries are executed on the reduced provenance graph using the

query API of the provenance repository. As the depth and size of the graph are reduced by multiple times compared to full provenance, both storage cost and the query complexity is reduced by several factors. The downside of reducing provenance on-the-fly is that once reduced full provenance cannot be recovered. As we present in Chapter 7, advantages of reduction out-weights the disadvantages when it comes to forward and backward provenance analysis.

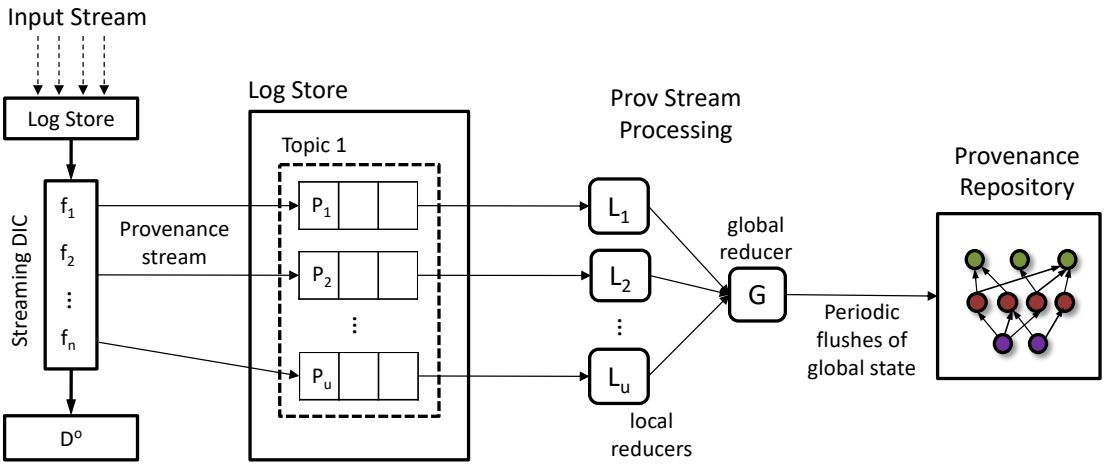


FIGURE 6.5. Stream processing solution for provenance generated by a streaming DIC

6.2.1. Processing infinite provenance streams. Handling infinite provenance streams is a special case for the streaming solution proposed above. Figure 7.3 shows the extended provenance stream processing model which supports infinite provenance streams. Infinite or long-running provenance streams are generated by stream processing DICs as they run for longer periods of time. The stream processing DIC is shown on the left of the figure and it consumes a data stream through a log store. The log store assigns a sequence number for

each stream element. Streaming DIC is again a set of functions and periodic analysis results (Ex: hourly statistics) are persisted in a storage system.

Provenance captured from the functions in Streaming DIC constructs the provenance stream that we are interested in. Same stream partitioning strategies (described below) are still applicable in this setup too. As the provenance stream is unbounded, Global reducer also maintains the reduced state only up to a configurable batch size (*global batch size*) and flushes it to the persistent provenance repository. Both degree of reduction and degree of order independence of the solution depends on the *global batch size*. For example, if an out-of-order element required for a reduction in *batch1* arrives in *batch2*, that reduction will not take place.

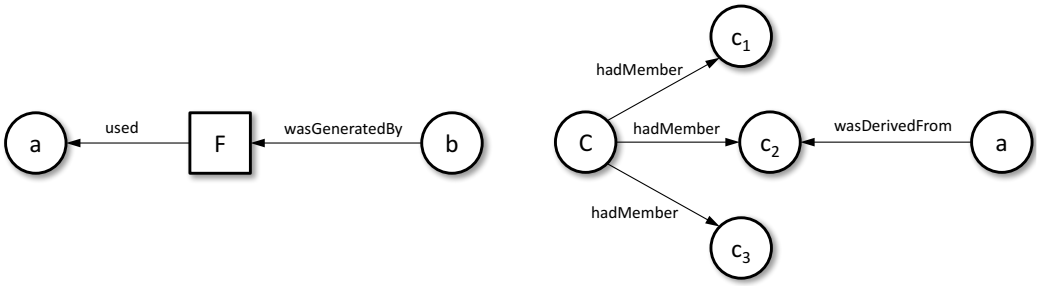


FIGURE 6.6. Possible derivation paths

6.2.2. Filtering irrelevant edges. The provenance stream definition given in Chapter 5 allows all types of W3C PROV edges as elements in the stream. When designing a streaming algorithm for forward and backward provenance, first we have to identify the set of edge types which can exist in a path between two entities. The streaming algorithm can only consider such edges and filter out the others. Direct derivation between two entities is represented by a *wasDerivedFrom* edge. As shown in Figure 6.6, a *used* edge and

a *wasGeneratedBy* edge connected through an activity node or a *hadMember* edge and a *wasDerivedFrom* edge connected through an entity node indicates a data derivation.

There are other W3C PROV edge types like *alternateOf*, *specializationOf* etc. too which may participate in derivation paths. However, for the purposes of this work in the context of DICs, we consider *wasDerivedFrom*, *used*, *wasGeneratedBy* and *hadMember* as the set of edge types that occur in a path between two entities. Other edge types we filter out. In addition, we further filter stream elements that add attributes to a node or an edge (second type in the definition) as those are not important for backward and forward provenance.

6.3. Parallel-prov-stream Algorithm

Our objective is an algorithm that can one-pass process provenance in parallel while adjusting for out-of-order events, and resulting in retention of backward and forward provenance. We illustrate this in action through Figure 6.7 which utilizes the full provenance graph given earlier in Figure 6.1. On the left is the full provenance graph for the computation which is streamed edge by edge as and when they are generated. The stream of full provenance is split into multiple partitions (using techniques in Section 6.4) and each partition is fed into a local reducer. On the right is the final reduced output from the global reducer which preserves backward and forward provenance.

Intermediate data items and edges are removed real-time by local and global stream processors. Both local and global processors maintain a state which contains the current set of reduced provenance edges. Each local stream processor filters out unnecessary edges first and processes only the selected W3C Prov edges as discussed in the previous section. New incoming edges are matched with the current local state to derive new dependencies

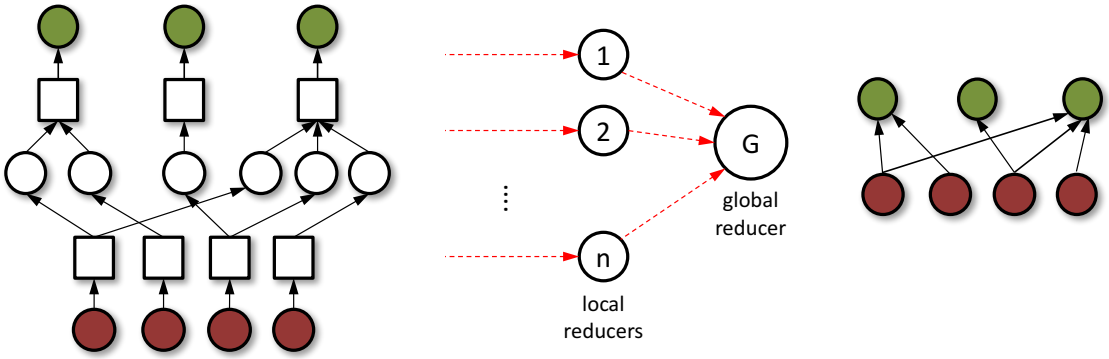


FIGURE 6.7. Application of parallel stream processing on a partitioned stream of full provenance to produce a reduced provenance graph preserving backward and forward provenance

by connecting them through common vertices. For example, if a local processor receives elements $\langle v_1, v_2 \rangle$, $\langle v_2, v_3 \rangle$ and $\langle v_5, v_6 \rangle$, it reduces the local state to $(\langle v_1, v_3 \rangle, \langle v_5, v_6 \rangle)$ by transitivity. Each local processor periodically flushes its reduced state into downstream global processor upon processing a configurable number (*local batch size*) of stream elements. The global processor further reduces the state by merging compatible edges from different local processors and it also periodically flushes the reduced state upon processing a fixed number (*global batch size*) of stream elements.

Algorithm 1 gives our one-pass *parallel-prov-stream* algorithm which is used by both local and global processors. This algorithm maintains an internal state which contains the current most reduced set of edges. Any snapshot of the internal state always satisfies the following condition.

For any edge $\langle v_i, v_j \rangle$ in the state, there is no other edge whose destination vertex is v_i or source vertex is v_j.

Algorithm 1 Provenance Stream Processing Algorithm

```
1: sMap                                              ▷ map of reduced edge lists by source id
2: dMap                                              ▷ map of reduced edge lists by destination id
3: procedure ADDEDGEGROUP(newEdges)                  ▷ newEdges: group of new stream elements
4:     list eDel                                     ▷ edges to delete
5:     list eAdd                                     ▷ edges to add
6:     for (ne in newEdges) do
7:         if (dMap.containsKey(ne.source)) then
8:             list edgesIntoSource = dMap.get(ne.source)
9:             for (e in edgesIntoSource) do
10:                eAdd.add(new Edge(e.source, ne.dest))
11:            end for
12:            eDel.addAll(edgesIntoSource)
13:        else if (sMap.containsKey(ne.dest)) then
14:            list edgesFromDest = sMap.get(ne.dest)
15:            for (e in edgesFromDest) do
16:                eAdd.add(new Edge(ne.source, e.dest))
17:            end for
18:            eDel.addAll(edgesFromDest)
19:        else
20:            INSERTEDGE(ne)                        ▷ add new edge into state
21:        end if
22:    end for
23:    if (!eAdd.isEmpty()) then
24:        ADDEDGEGROUP(eAdd)                        ▷ further reductions
25:    end if
26:    for (edge in eDel) do
27:        DELETEEDGE(edge)                          ▷ delete edge from state
28:    end for
29: end procedure
30: procedure ADDEDGE(newEdge)                       ▷ newEdge: new stream element
31:    list newEdges
32:    newEdges.add(newEdge)
33:    ADDEDGEGROUP(newEdges)
34: end procedure
35: procedure INSERTEDGE(edge)▷ inserts entries into sMap and dMap for given new edge
36: end procedure
37: procedure DELETEEDGE(edge)          ▷ delete entries in sMap and dMap for given edge
38: end procedure
```

The algorithm maintains two Map data structures *sMap* and *dMap* for efficient access to edges in the state. Map *sMap* is a collection of key-value pairs where the key is a vertex id and value is a list of edges whose source is the same vertex id. Map *dMap* is a collection of key-value pairs where the key is a vertex id and value is a list of edges whose destination is the same vertex id. A given edge has two pointers from *sMap* and *dMap* based on its source vertex id and destination vertex id. For each new stream element or edge, the internal state is checked to find possible reductions. The algorithm uses the two Maps to access the edges with possible reductions in $O(1)$ time without scanning through the entire state.

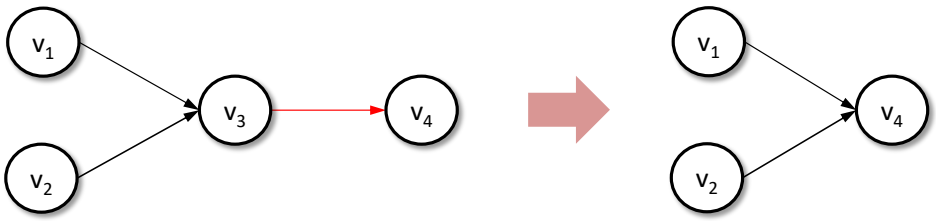


FIGURE 6.8. Reduction through the source vertex of a new edge

Figure 6.8 shows how a reduction through the source vertex of a new edge happens (line number 7 to 12 in Algorithm 1). Edges $\langle v_1, v_3 \rangle$ and $\langle v_2, v_3 \rangle$ are already present in the local state and $\langle v_3, v_4 \rangle$ is the new edge. When the algorithm calculates the new state, $\langle v_1, v_3 \rangle$ and $\langle v_2, v_3 \rangle$ are added to the list of edges to be deleted (*eDel*), and $\langle v_1, v_4 \rangle$ and $\langle v_2, v_4 \rangle$ are added to the list of new edges to be added (*eAdd*). Now the new edges $\langle v_1, v_4 \rangle$ and $\langle v_2, v_4 \rangle$ may participate in further reductions if there are edges with v_4 as the source vertex in the local state. The algorithm uses recursion (line 24) to compute the full list of new edges before adding them into the local state. Reduction through the destination vertex (line number 13 to 18 in Algorithm 1) also works the same way.

There are two operations to process a single stream element (*addEdge*) and a group of stream elements together (*addEdgeGroup*). This algorithm does not depend on the order of provenance edges in the stream within the configured *batch size* as it keeps unreduced edges in the state for future reductions. Space complexity of the algorithm is bounded by the *local batch size* for local reduction and *global batch size* for global reduction.

6.4. Partitioning a Provenance Stream

In order to handle high rates of provenance from large-scale DICs, the streaming system should be scalable. As shown in Figure 6.7, we split the stream of provenance into partitions and process them in parallel. Partitioning leads to seamless horizontal scalability of the system. Locally processed results from parallel stream processors are periodically merged to compute the current global state of backward and forward provenance. The partitioning strategy is extremely important for the efficiency of the system.

As we discussed above, the provenance stream that we focus is always a graph stream. Our stream processing algorithm transitively merges provenance graph edges through common vertices. When processing a partitioned provenance stream, the algorithm performs local reductions only within the assigned partition. The efficiency of the streaming system increases when the number of local reductions within each reducer increases. That leads to smaller state size (memory footprint) within local reducers, less processing time for new stream elements and faster serialization towards the global reducer. All these factors contribute to higher throughput.

Degree of local reduction within a partition mainly depends on the partitioning strategy. In order to maximize local reduction, provenance stream should be partitioned so that the

edges sharing the same vertices fall into the same partition as much as possible. When such edges are distributed across partitions, their reductions do not happen at the local reducer level and the number of edges reaching the global reducer increases. That reduces the advantage of using parallel stream processing.

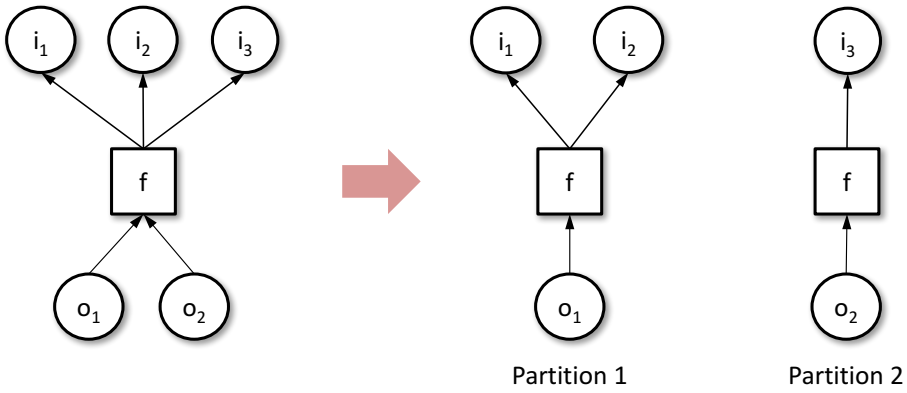


FIGURE 6.9. Partitioning provenance from a function execution

We evaluate three different partitioning strategies. In order to not lose dependency paths during the partitioning process, we introduce a constraint that applies for all partitioning strategies:

All provenance edges generated during a single function execution must belong to the same partition in the stream.

A function in a DIC is executed many times on small fractions of input data and here we focus on such single execution of a function. Consider the scenario in Figure 6.9 where function execution f consumes three inputs (i_1, i_2 and i_3) and produces two outputs (o_1 and o_2). Suppose provenance from this function execution is split into two partitions as shown in the diagram. The reduced output, then, from the first partition is ($\langle o_1, i_1 \rangle$, $\langle o_1, i_2 \rangle$) and from the second partition is ($\langle o_2, i_3 \rangle$). These two outputs are received by the global

reducer which outputs ($\langle o_1, i_1 \rangle$, $\langle o_1, i_2 \rangle$, $\langle o_2, i_3 \rangle$). This output is incorrect as it lacks three valid dependency paths $\langle o_2, i_1 \rangle$, $\langle o_2, i_2 \rangle$ and $\langle o_1, i_3 \rangle$. Above constraint avoids this issue by sending all provenance edges from a single function execution to the same partition.

We consider three different partitioning strategies and evaluate their performance in the context of DICs. In all three strategies, the smallest non-separable unit for partitioning the provenance stream is a collection of edges from a single function execution.

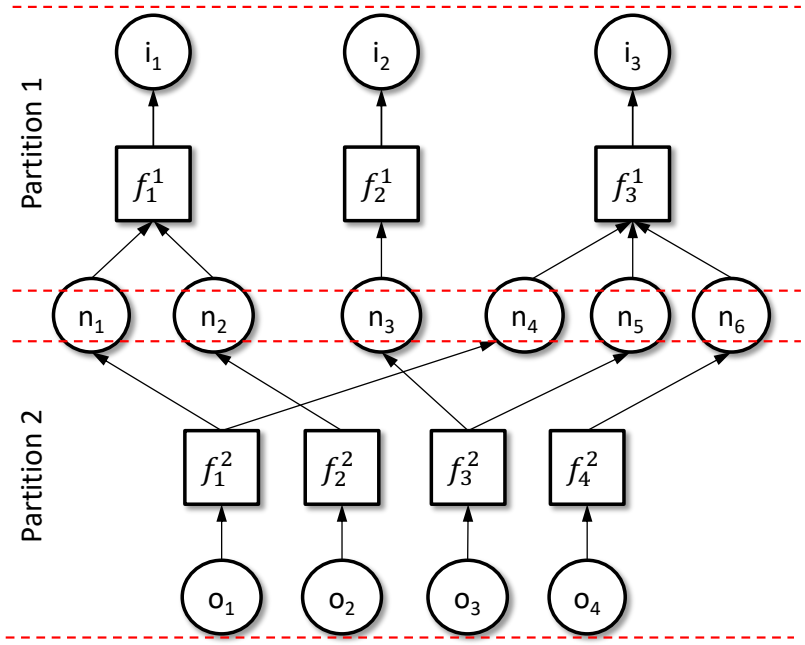


FIGURE 6.10. Horizontal partitioning

Horizontal Partitioning: Provenance elements from each function (all its function executions) in the DIC are directed to a separate partition. Each partition could only perform a one-step reduction which creates dependencies between the inputs and outputs of the relevant function. Horizontal partitioning for a stream of provenance from a DIC which consists of two functions is shown in Figure 6.10 in which provenance elements from functions f^1 and f^2 will be processed by separate local stream processors. Provenance from

function execution f_1^1 belongs to *partition 1* and will be reduced to ($\langle n_1, i_1 \rangle$, $\langle n_2, i_1 \rangle$). Further reductions are not possible as the adjacent edges $\langle f_1^2, n_1 \rangle$ and $\langle f_2^2, n_2 \rangle$ belong to *partition 2*. Uneven load distribution among partitions can be expected with horizontal partitioning as different functions deal with different sizes of input data. In addition to that, horizontal partitioning can lead to poor scalability as the number of partitions can not grow beyond number of functions used in the DIC.

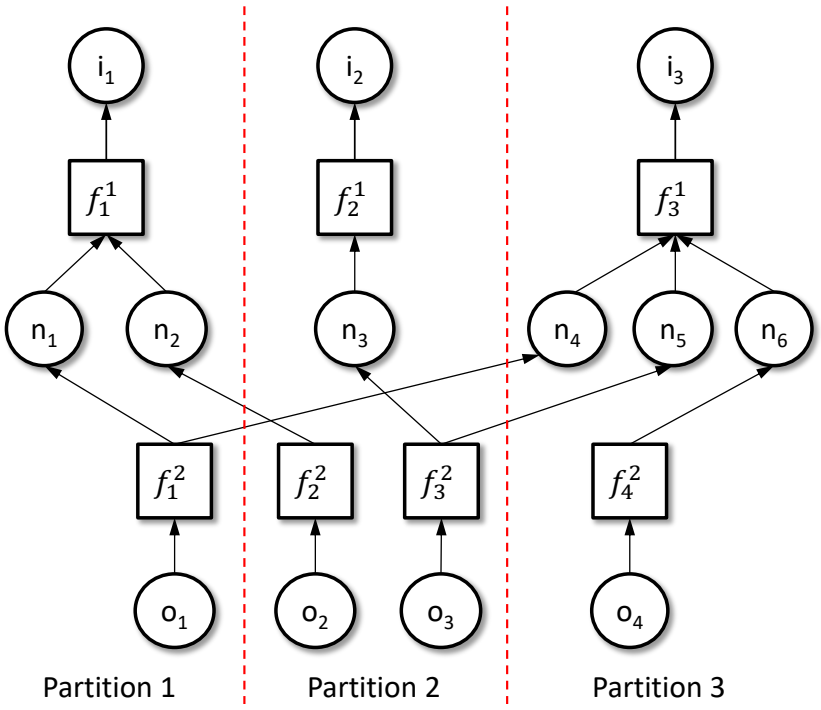


FIGURE 6.11. Vertical partitioning

Vertical Partitioning: Provenance stream is partitioned vertically along the derivation paths between inputs and outputs. Figure 6.11 shows the vertical partitioning for the same example in Figure 6.10. The idea is to preserve derivation paths within partitions as much as possible and maximize local reduction.

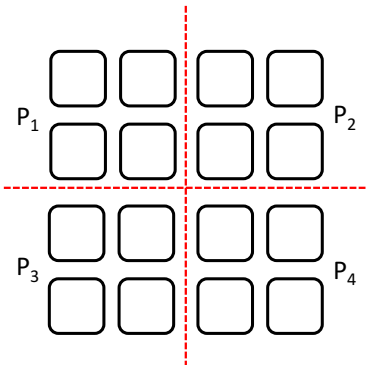


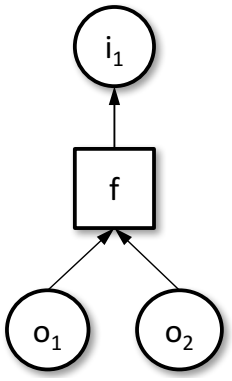
FIGURE 6.12. A cluster partitioned based on locality

DIC frameworks like Hadoop and Spark consider data locality as a major factor when scheduling functions on slave nodes. For a group of cluster nodes located close to each other, there is a high chance that a higher percentage of functions processing the data stored on them are executed within themselves. For example in Hadoop, nodes to run *map* functions are selected by the Resource Manager as near as possible to the Input block locations. Outputs from *map* functions are stored in same nodes. Then the *reduce* functions are also executed considering the locality of *map* outputs. Therefore, we propose to partition the stream based the node which generated each stream element. The cluster of nodes is partitioned based on the locality as shown in Figure 6.12 and provenance stream elements from each cluster partition create a separate partition in the provenance stream.

Random Partitioning: Randomly distributes provenance from function executions among parallel local stream reducers. Degree of local reduction depends on the percentage of provenance from nearby function executions which goes into the same partition. When the number of partitions increases, performance of random partitioning is expected to decrease as the probability of reducible edges falling into the same partition decreases. If

the number of partitions is small random partitioning may perform better than horizontal partitioning.

6.5. Early Elimination Problem

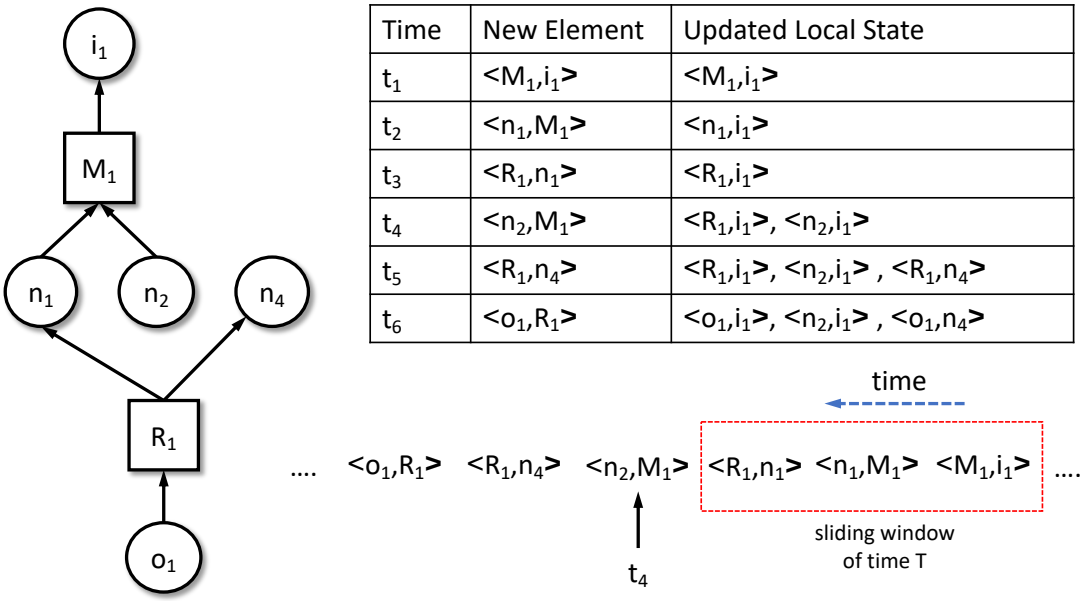


Time	New Element	Updated Local State
t_1	<f,i_1>	<f,i_1>
t_2	<o_1,f>	<o_1,i_1>
t_3	<o_2,f>	<o_1,i_1>, <o_2,f>

FIGURE 6.13. Early elimination example

In our *parallel-prov-stream* algorithm, one vertex is permanently removed during each reduction. When edges $\langle v_i, v_j \rangle$ and $\langle v_j, v_k \rangle$ are reduced to $\langle v_i, v_k \rangle$, vertex v_j is removed and no longer available for further reductions. This leads to incorrect results if v_j participates in other edges which have not been received by the processor yet. This situation is possible only with activity nodes as an intermediate data item (entity) in a computation can only be used by one function. We call this as the *early elimination problem*. Consider the scenario in Figure 6.13 in which a function uses a single input and generates two outputs. At time t_2, the local state is reduced to ($\langle o_1, i_1 \rangle$) by removing f. When $\langle o_2, f \rangle$ arrives at t_3, there is no way to derive its dependency on i_1. We evaluate two strategies to avoid this problem.

Sliding Window: Each stream processor maintains a sliding window of a configurable time which retains a limited number of past stream elements. Both the internal state and the sliding window is checked (an extension to above algorithm) at the arrival of each element



Time	New Element	Updated Local State
t_1	$\langle M_1, i_1 \rangle$	$\langle M_1, i_1 \rangle$
t_2	$\langle n_1, M_1 \rangle$	$\langle n_1, i_1 \rangle$
t_3	$\langle R_1, n_1 \rangle$	$\langle R_1, i_1 \rangle$
t_4	$\langle n_2, M_1 \rangle$	$\langle R_1, i_1 \rangle$, $\langle n_2, i_1 \rangle$
t_5	$\langle R_1, n_4 \rangle$	$\langle R_1, i_1 \rangle$, $\langle n_2, i_1 \rangle$, $\langle R_1, n_4 \rangle$
t_6	$\langle o_1, R_1 \rangle$	$\langle o_1, i_1 \rangle$, $\langle n_2, i_1 \rangle$, $\langle o_1, n_4 \rangle$

FIGURE 6.14. Sliding window solution for the early elimination problem

to compute the new local state. If a certain deleted element in the local state is not found in the sliding window too, dependencies will be lost and the final result will not be 100% accurate.

In the example shown in Figure 6.14, reduction at t_2 removes M_1 from the local state. Now when $\langle n_2, M_1 \rangle$ arrives at t_4, derivation $\langle n_2, i_1 \rangle$ can not be computed using the local state as it does not include M_1. The figure shows how a sliding window solution is used to solve the *early elimination problem*.

Time	New Element	Updated Local State
t_1	$\langle M_1, i_1 \rangle$	$\langle M_1, i_1 \rangle$
t_2	$\langle R_1, n_1 \rangle$, $\langle R_1, n_4 \rangle$	$\langle M_1, i_1 \rangle$, $\langle R_1, n_1 \rangle$, $\langle R_1, n_4 \rangle$
t_3	$\langle n_1, M_1 \rangle$, $\langle n_2, M_1 \rangle$	$\langle R_1, i_1 \rangle$, $\langle n_2, i_1 \rangle$, $\langle R_1, n_4 \rangle$
t_4	$\langle o_1, R_1 \rangle$	$\langle o_1, i_1 \rangle$, $\langle n_2, i_1 \rangle$, $\langle o_1, n_4 \rangle$

FIGURE 6.15. Reduction with Grouped Usages and Generations

Grouping: When multiple data items are used or generated by a function, the provenance collector sends all usage or generation edges as a single group of elements in the stream. The *parallel-prov-stream* algorithm processes the group together through *addEdge-Group* operation which makes sure that the vertices are deleted only after considering all elements in the group for reductions. In this case, the reduction for the same example happens as shown in Figure 6.15. This technique works only if the stream producer can be controlled according to the requirements of the provenance stream processing system. If the stream processing solution is applied on a provenance stream produced by some third party application, Grouping technique is not applicable and the Sliding Window technique is the only option.

6.6. PID based Provenance Identity in Data Lake

Our provenance stream processing system is based on a distributed log store which facilitates as a short term storage layer for data streams. Log stores like Kafka supports "Topics" which uniquely identifies a single stream. Topics, and their uniqueness, is used to guarantee that provenance events are associated with the correct DIC from which they were generated. At the start of a DIC, it is assumed to know the unique topic ID to which it should publish provenance events. Figure 6.16 shows an extension for the provenance stream processing system to introduce components which orchestrates DIC workflows within a Data Lake environment.

The "Data Lake Job Manager" is responsible for coordinating each DIC and its provenance stream analysis process. When a request for a new DIC comes in, the Job Manager registers DIC metadata in the "Data Lake Metadata Store" and mints a new persistent

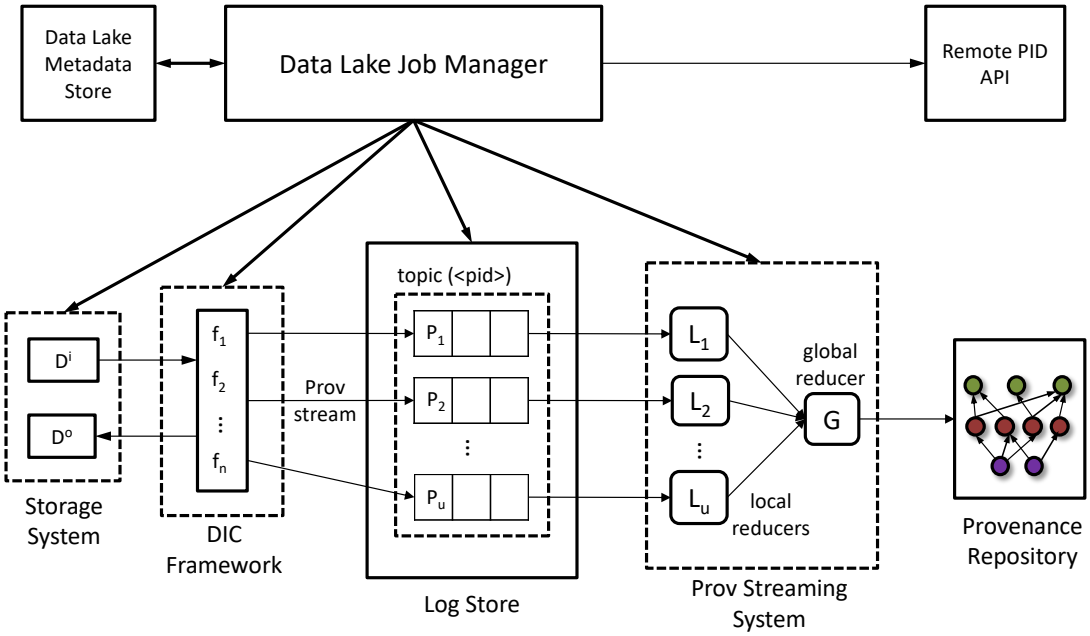


FIGURE 6.16. Extended streaming solution based on PIDs to handle provenance identity, reproducibility and archival

identifier (PID [**106**]) by calling the remote PID API, associating metadata from Metadata Store. When the PID is resolved, it downloads the metadata record of the DIC execution. This metadata includes input and output dataset paths in the storage layer, configuration file location, start date/time, software versions etc. After registering metadata, Job Manager creates a topic using the minted PID to store the provenance stream from DIC. As the next step, the Job Manager calls the provenance streaming system to start the stream consumer to read from the created topic in the log store. Streaming system is configured to write reduced provenance outputs to the provenance repository using the same PID. Finally, it includes the minted PID as a parameter in the DIC input configuration and calls the DIC framework to start the DIC. Job manager makes sure that all components of the environment learn the

unique PID assigned for each DIC and updates the metadata store so that it can be queried to find details about previously executed DICs.

The root PID minted above can be updated with a reference to child PIDs to publish results from the provenance streaming system. For batch DICs, final reduced provenance graph is exported from the Provenance Repository and a new PID is minted for that. Now the root PID for the DIC is updated by adding the new PID as a child. This child PID can be used to download the reduced provenance graph. For Streaming DICs, as the reduced provenance graph is unbounded, daily or hourly results can be published by minting child PIDs.

This proposed PID based solution helps in reproducing the provenance stream analysis. Metadata record associated with the root PID can be used to find configurations related to DIC, log storage, and the streaming system. Reproducibility can be further increased by including Git repository URLs related to provenance capturing and stream processing implementations. One constraint related to reproducibility of Data Lake experiments is that datasets from most Data Lakes are not exported out of the Data Lake due to their volumes and challenges in managing them. Therefore reproducibility is mostly limited to the same Data Lake environment.

CHAPTER 7

Implementation and Evaluation

Here we evaluate our parallel provenance stream processing techniques discussed above using a prototype implementation of the proposed stream processing architecture. We use DIC frameworks from the Apache Big Data stack to implement multiple use cases. Hadoop [**16**] and Spark [**108**] are used to implement our use case applications. Kafka [**58**] is used as the distributed log store. Our streaming solution based on the *parallel-prov-stream* algorithm is implemented using Flink Streaming [**26**] framework. We evaluate our techniques for both finite (from a batch processing DIC) and infinite (from a stream processing DIC) provenance streams. We evaluate the accuracy of the reduced provenance graph, the throughput of the system and degree of local reductions for different partitioning strategies and different methods to mitigate *early elimination problem*.

7.1. Parallel provenance stream implementation

Our goal is to evaluate our streaming solution for both finite and infinite provenance streams. In order to generate a finite provenance stream, we use a batch processing DIC which terminates after processing a stored data set of a fixed size. An infinite provenance stream is generated using a stream processing DIC. Both use cases are implemented using Twitter data.

7.1.1. Finite provenance stream implementation. We use a workflow consisting of two batch processing DICs to process Twitter data and apply our parallel provenance stream processing technique on provenance generated by the DICs. As shown in Figure 7.1, a Twitter client is used to collect tweets through the Twitter public streaming API and store in HDFS over a period of time. For each tweet, the client captures the Twitter handle of the author, time, language and the full message and writes a record into an HDFS file. First DIC which is implemented using Hadoop (v2.8.1) counts the occurrences of each hashtag in the full Twitter dataset and writes the results into a new HDFS file. The second DIC in the workflow is implemented using Spark (v2.2.1) and it produces aggregated tweet counts according to categories (sports, movies, politics etc).

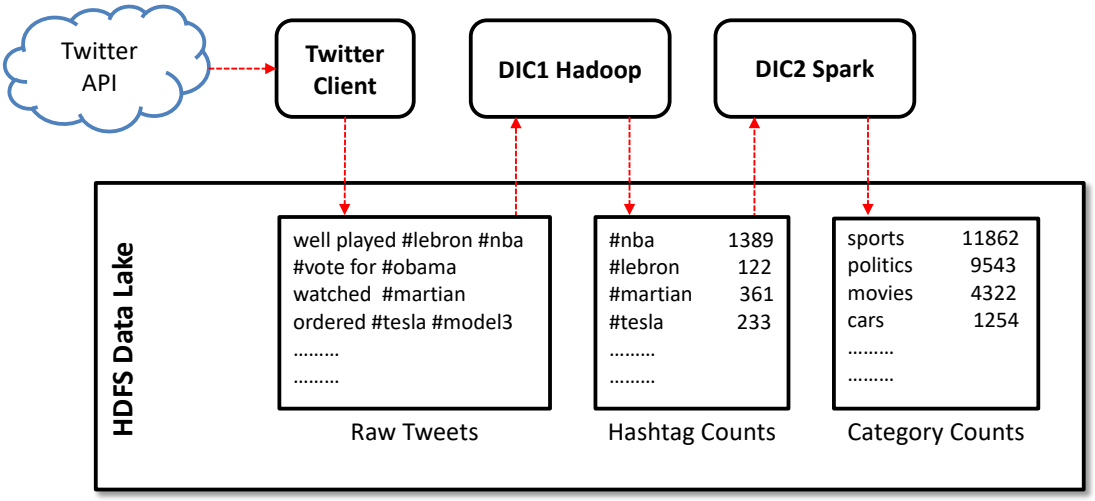


FIGURE 7.1. DIC workflow to categorize hash tags in Twitter data

We implement our *parallel-prov-stream* algorithm on top of the Flink Streaming framework (v1.6.0) **[26]** since Flink provides support for stateful stream processing while producing high throughput and low latency. We employ the Kafka (v0.11.0.1) **[58]** distributed log store to persist the provenance streams generated from DICs in the workflow and to handle

partitioning. Kafka retains stream elements for a configurable period of time (7 days by default) and controls the data rate going into the streaming system. Flink provides built-in Kafka connectors which can be configured to pull stream elements from Kafka partitions into Flink consumers.

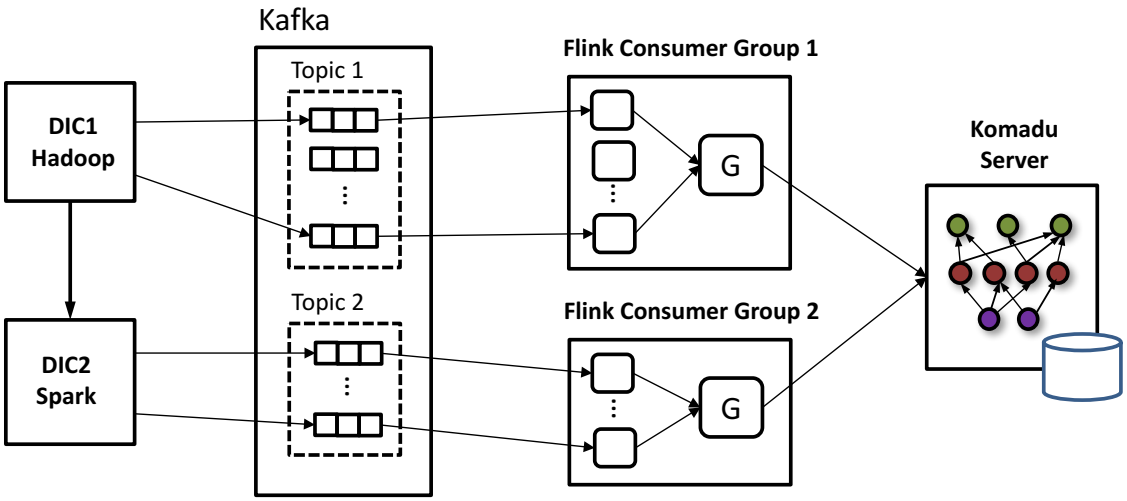


FIGURE 7.2. Provenance stream processing architecture for a batch processing DIC workflow

The streaming solution is applied for provenance streams from each DIC in the workflow. Figure 7.2 shows the overall architecture of the system for the above DIC workflow. The functions in Hadoop and Spark jobs are instrumented to capture provenance in W3C PROV JSON format. The Kafka producer API is used to write streams into Kafka. Kafka producer batches stream elements to reduce the network overhead of calling Kafka API for every single stream element. We use a batch size of 50000 and a memory buffer size of 1GB for all our experiments.

There are different approaches such as instrumenting, log parsing, wrapping or extending frameworks etc. [53] [5] to capture provenance from computations. There are different

trade-offs [**89**] associated with them depending on the type of system. Our streaming solution is independent of the provenance capture method. We use instrumentation for ease of implementation.

Kafka supports the concept of Consumer Groups where all consumers in a group are reading from the same Topic. Each partition in a Topic is read by only one consumer in a Consumer Group. As shown in Figure 7.2, we use two different streaming jobs deployed in Flink which act as separate Kafka consumer groups. These two consumer groups consume provenance streams from DIC1 and DIC2. The number of consumers in each streaming job is set to the number of partitions in the provenance stream. Each partition consumer in Flink is considered as a *local reducer* described in Chapter 6.

The Hadoop job consists of *Map*, *Combine* and *Reduce* functions while the Spark job consists of *MapToPair* and *ReduceByKey* functions. Horizontal partitioning is done by assigning a separate partition for each function in a DIC. Vertical partitioning is done by assigning a separate partition for a subset of nodes in the Hadoop or Spark cluster. Each DIC is assigned a separate Kafka topic and each partition in the provenance stream maps to a separate partition in the Kafka topic. Both local and global processors in Flink implement the *parallel-prov-stream* algorithm. Each local steam processor consumes a single partition from Kafka and performs local reduction. Local reducers periodically emit the reduced results into the global reducer which further reduces the global state.

As the provenance streams from batch DICs are finite, we emit final reduced graph only at the end of the provenance stream. If no new stream elements are received for a configurable time limit, the global reducer deduces end of stream is reached and completes

the output reduced provenance graph. This reduced graph is then stored in Komadu central provenance repository. When the DIC workflow continues to run, reduced provenance graphs from all DICs are stored in Komadu and merged together using unique identifiers assigned for data items. Backward and forward provenance for entire workflow is derived by merged reduced graphs from all DICs.

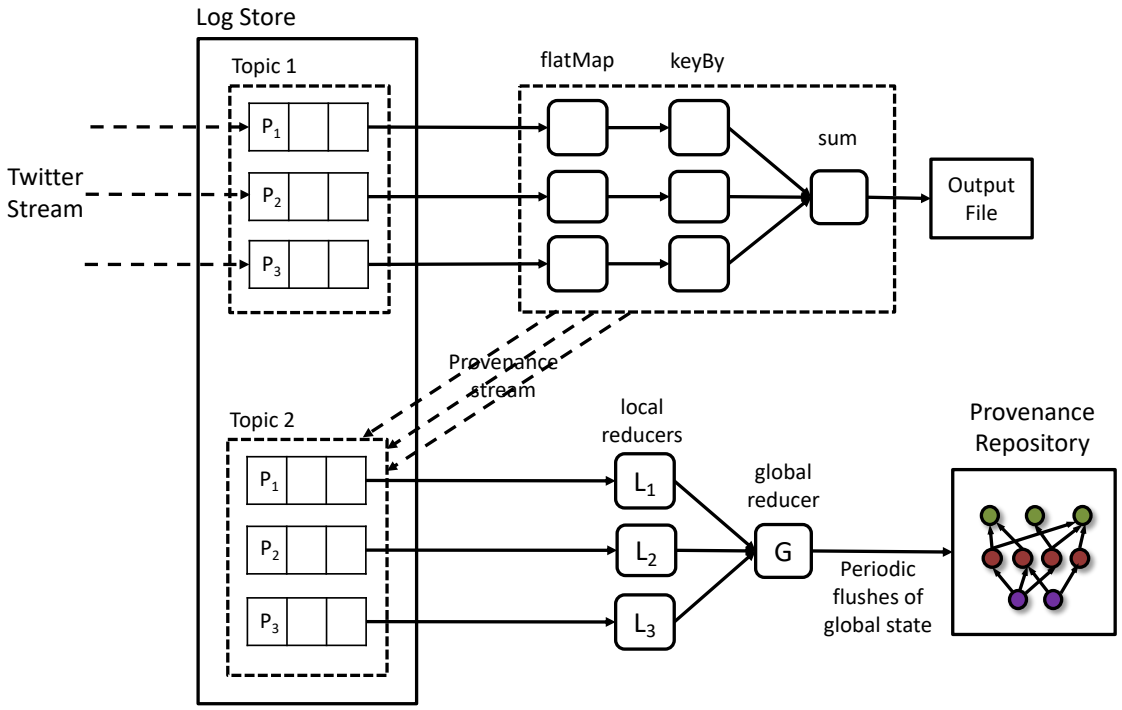


FIGURE 7.3. Provenance stream processing architecture for a stream processing DIC.

7.1.2. Infinite provenance stream implementation. To generate an infinite provenance stream, we implement a stream processing DIC for the same use case shown in Figure 7.1. Instead of using batch DICs in Hadoop and Spark, we use a streaming job implemented in Flink to count hashtags first and then categorize them. Figure 7.3 shows the provenance stream processing architecture in this case. First, the Twitter stream is written to the Kafka

log store and read by the streaming system for hashtag analysis. Note that the figure shows only the hashtag counting part of the streaming job. Provenance stream is generated by the instrumented functions *flatMap*, *keyBy* and *sum*. Then the provenance stream is written to a separate topic in Kafka and processed by the local and global reducers. As the provenance stream will be infinite, global reducer emits reduced graph partitions periodically.

Generated provenance is also considered another parallel stream within the hashtag processing system. Figure 7.4 shows how both provenance and hashtags from *flatMap* function are treated as a single stream and then it is split into two separate streams. This is applied to all functions and the provenance stream is written back to a Kafka topic as a sink. Provenance from one parallel Flink consumer is written to a single partition in the provenance stream. That leads to vertical partitioning of the provenance stream.

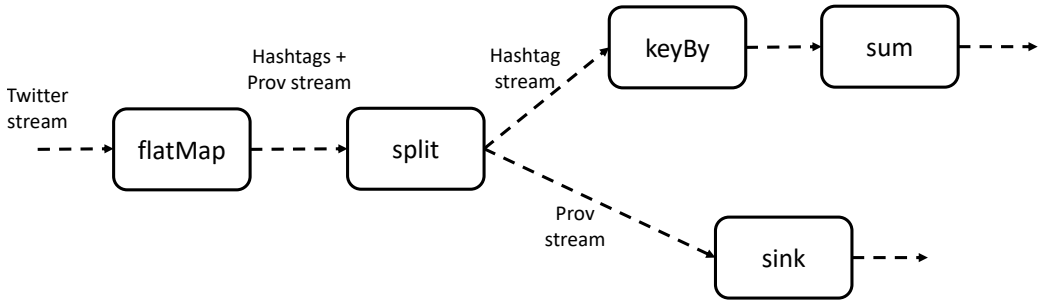


FIGURE 7.4. Provenance from a streaming function is also considered as a parallel stream

Kafka assigns an identifier called the ”offset” for each record stored. It is used to track the position of the record within a partition. During the process of capturing provenance, we use the offset of each record as the unique identifier. Identifiers for intermediate data products emitted by the functions are generated using UUIDs. Provenance collected from

a streaming job is only useful within a certain time window. A log store retains original stream elements only for its retention period. Backward and forward provenance is used to analyze the relationships between input and output data. Often, the output for a streaming job is a time-bounded summary of the analysis. For example, daily or hourly reports or dashboard updates. Therefore, backward and forward provenance are useful only within the retention period of the original stream elements. An example use case is a situation where some abnormal result is shown by an hourly result of a streaming job. Backward provenance can be used to trace the original stream data in the log store within its retention period. If the full original stream is persisted to a permanent storage for post analysis, provenance can be used for a longer period of time.

7.2. Experimental Evaluation

We evaluate our streaming solutions against both finite and infinite provenance streams using the implementations discussed above. Evaluation is focused on the accuracy of the results, throughput of the system, degree of reductions performed and query performance. Experiments are run on virtual machines from the Jetstream cloud environment **[41]**. We use medium size VMs which consist of 6 CPU cores of 2.5 GHz speed, 16 GB of RAM and 60 GB of disk space per instance.

7.2.1. Finite provenance stream evaluation. As discussed above, we use a batch processing DIC workflow (see Figure 7.2) to evaluate the streaming solution against a finite provenance stream. This evaluation includes multiple experiments. First, we evaluate the accuracy of our provenance stream processing algorithm for out-of-order provenance

streams. We use a simulator to produce out of order streams and measure the accuracy for both "Grouping" and "Sliding Window" methods for mitigating *early elimination problem*. Second, the efficiency of the three partitioning strategies is evaluated by measuring the degree of local reduction. Third, we evaluate the scalability of our streaming solution under increased parallelism. Finally, we evaluate the query performance for backward and forward provenance on full provenance vs reduced provenance.

Environment. The HDFS cluster consists of 17 nodes with one master and 16 slave nodes. Total of 10.1 GB Twitter data was collected and stored on HDFS. Four nodes are allocated for the Kafka cluster where the master node runs the zookeeper instance and all four nodes run Kafka brokers. Up to nine nodes are used for Flink streaming cluster depending on the experiment where one node is always used as the master and all others acting as slaves. Another VM is allocated for Komadu which persists reduced provenance graphs coming out of the streaming system.

Workload. The workload is a DIC workflow composed of two DICs: *DIC1* runs on Hadoop and *DIC2* runs on Spark, see Figure 7.2. Each DIC produces a separate provenance stream and each is processed in isolation. The provenance results, once reduced to backward and forward provenance for both DICs, are brought together in Komadu [**93**] to build workflow level provenance. The first three experiments below present results based on the provenance stream from *DIC1* as it generates a larger volume of provenance compared to *DIC2*. The last experiment on query performance uses reduced provenance from both DICs.

In the first experiment, we measure the accuracy of our *parallel-prov-stream* algorithm for out-of-order provenance streams and compare results with Chen et al. [**28**]. Accuracy

is measured by calculating the percentage of correct backward and forward provenance relationships exist in the output provenance graph. The algorithm in [**28**] is not designed to handle out-of-order events. Therefore, the intention of this experiment is not to show that our algorithm performs better. But we use their algorithm as a base case to show the importance of handling out-of-order events for accuracy.

As [**28**] is not a parallel algorithm, the provenance stream from *DIC1* is considered as a single partition and consumed by only one stream processor in this experiment. Same experiment is executed against both algorithms to measure the accuracy of the final results against varying out-of-order levels. In order to control the ingress throughput going into the stream processor we had to first record provenance from *DIC1* and then replay it at a certain rate. As we need to vary the percentage of out-of-order elements in the stream, we developed a simulator tool which produces streams with required levels of ordering errors, consuming the recorded file with correctly ordered provenance events. A subset of 1.1 GB input data was used for this experiment and it generated 2.86 GB of provenance.

We measure the accuracy of both "Grouping" and "Sliding Window" (window size set to 10 seconds) approaches of avoiding *early elimination problem*. According to results shown in Figure 7.5, all three algorithms provide 100% accuracy for perfectly ordered streams (0% out-of-order elements). As the fraction of out-of-order elements increases, our algorithm remains 100% accurate with "Grouping" method as expected. That is because all provenance edges related to a single function execution is grouped and processed together as a single stream element. The "Sliding Window" method shows some inaccuracy with increasing ordering errors due to missed reductions related to edges older than the window

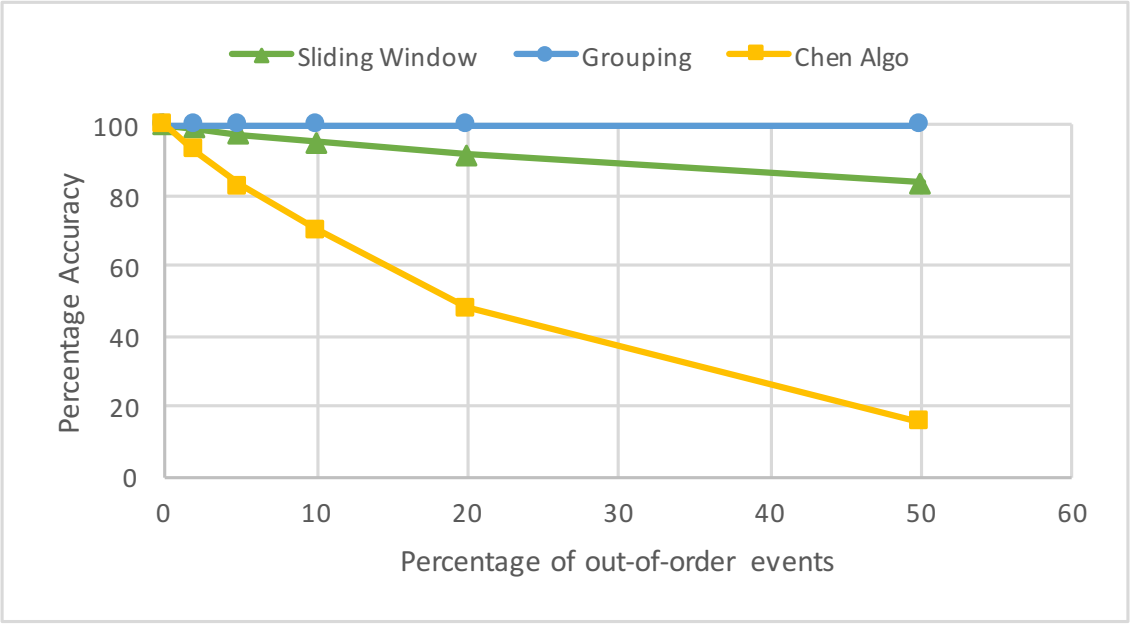


FIGURE 7.5. Percentage accuracy of backward and forward provenance results against the percentage of out-of-order elements for "Grouping" and "Sliding Window" methods of our algorithm and Chen algorithm

size. The accuracy of Chen's algorithm reduces considerably as it depends on the strict order of stream elements. We use "Grouping" method in all remaining experiments as it provides the best accuracy.

In the second experiment, we evaluate the efficiency of horizontal, vertical and random partitioning strategies by measuring the degree of local reduction for the same computation under the same range of local batch size. The stream partitioning strategy and local batch size are both determinant factors affecting the degree of local reduction. The degree of local reduction is measured by counting the total number of edges emitted by parallel local reducers towards the global reducer during the entire computation. We use the full 10.1 GB dataset for this experiment and run 16 Hadoop slaves and 8 Flink slaves.

Vertical partitioning performs best as it leads to maximum reduction along derivation paths, see Figure 7.6. We considered provenance from a single Hadoop node as a single stream partition. All functions executed within a single node contribute to reductions within the same partition. Random partitioning also shows better reduction with increasing local batch size. This is due to the increasing probability of provenance from adjacent function executions falling under the same partition. However, the degree of reduction is less compared to vertical partitioning. Horizontal partitioning shows almost constant reduction with varying local batch size. This is expected as a horizontal partition can only have provenance from function executions of a single function. Reductions across multiple function executions are not possible and the chance of reductions does not increase with the batch size.

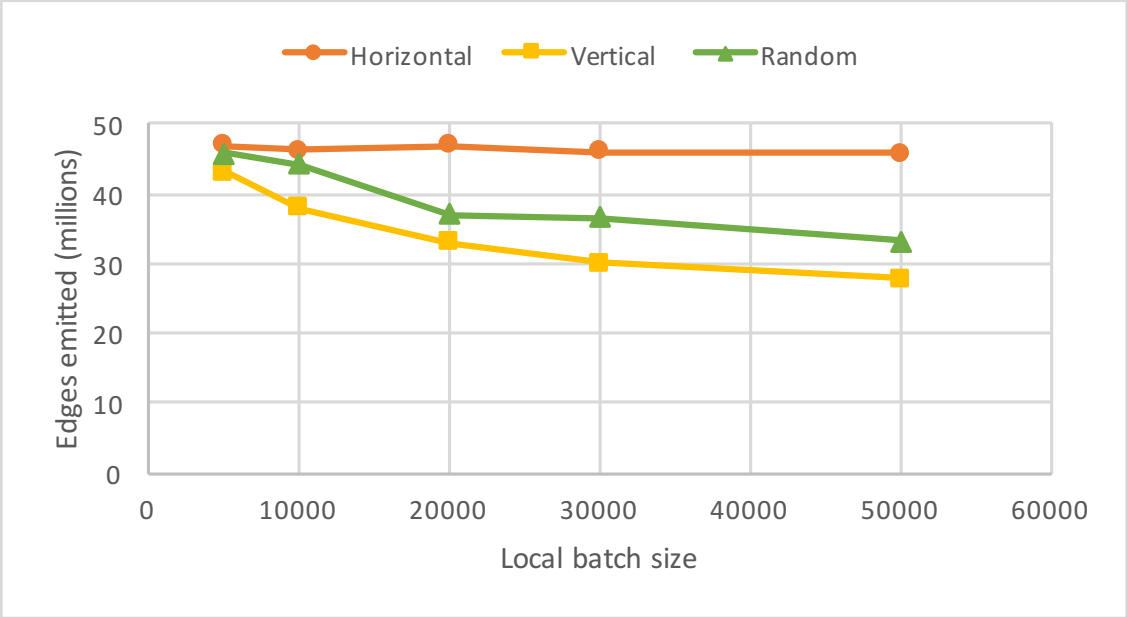


FIGURE 7.6. Number of edges emitted by local reducers against the *local batch size* (in number of elements) for horizontal, vertical and random partitioning strategies

The third experiment evaluates the scalability of the system through speedup: the proportional reduction in execution time for increasing parallelism against a fixed load. The dataset used is the full 10.1 GB Twitter dataset; the local batch size is set to 20000. Parallelism is increased from 1 to 16 by employing the same number of nodes in the Hadoop Cluster. For each level of parallelism, the same number of Kafka partitions and parallel Flink consumers are created. Vertical partitioning is used as the partitioning strategy where provenance from each node in the Hadoop cluster contributes to a single partition.

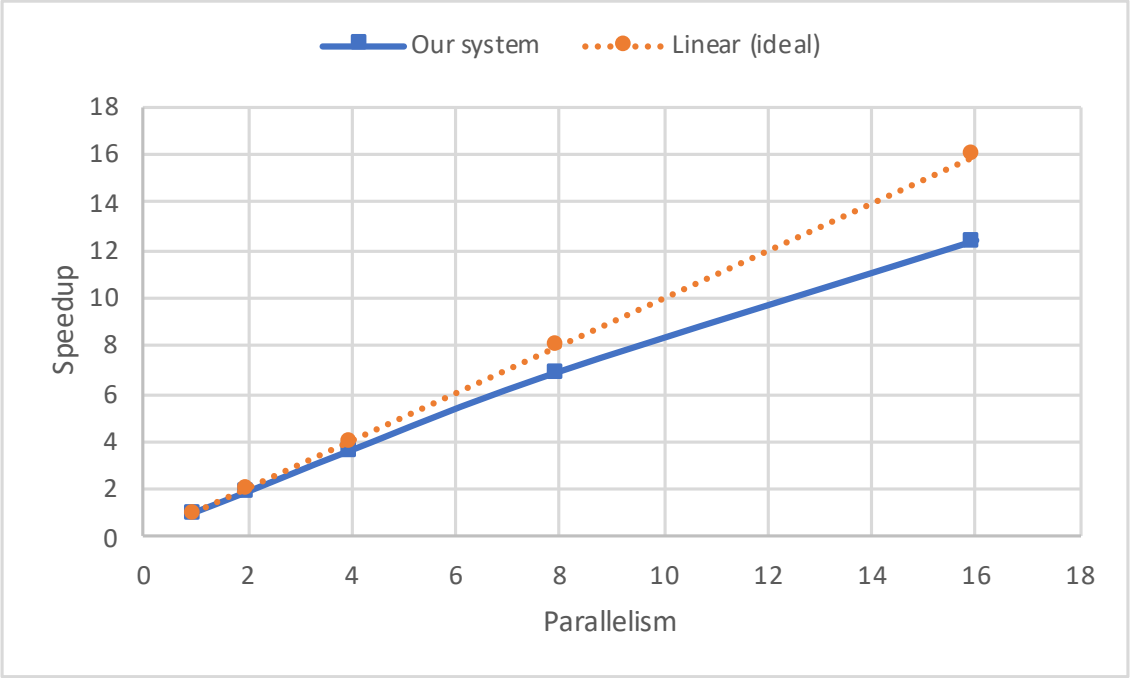


FIGURE 7.7. Speedup (parallelism = 1 time/parallelism = n time) of the system against increasing parallelism. Input data size is fixed at 10.1 GB and local batch size is 20000.

Our approach shows sub-linear speedup with the increasing parallelism, as shown in Figure 7.7. The system shows a maximum throughput of 6.044 MB/s per partition during this experiment. The speedup deviates from the ideal line with increasing parallelism.

TABLE 7.1. Execution times for provenance queries

Provenance Query	Full Provenance	Reduced Provenance
backward-provenance(”sports:4341”)	26.8s	2.3s
forward-provenance(”.. #a #b #c ..”)	1.2s	0.85s

There are multiple factors contributing to this deviation. As we discussed above, the global reducer becomes a bottleneck when the aggregate output throughput from local reducers exceeds the capacity of the global reducer. At this point, the streaming system should be further scaled using a multilevel topology. Other factors are, cross partition edges which do not allow 100% reduction within a partition and possible communication overheads in the streaming system with increasing parallelism.

As the parallelism increases, if the ingress throughput to the global reducer is too much, global reducer becomes a bottleneck. Therefore, at that point, global reducer should also be scaled-up. Having less number (ideally one) of global reducers provides the best results (most compressed output graph) as it leads to maximum reductions within the same state. However, when the single global reducer becomes a bottleneck for scalability, it must be scaled up. During our speedup experiment, global reducer showed constant throughput and didn't show signs of regression up to the maximum parallelism of 16.

Table 7.1 shows the results for query execution times on full provenance against reduced provenance. Queries are executed using the Komadu query API. First, we selected a backward provenance query for an output data product (”sports:4341”) which is derived by a high number of input tweets. When the backward query is executed on the full provenance graph, backward graph traversal happens through multiple functions in DIC1 and DIC2. However, the depth of graph traversal in the reduced graph is limited to two. As

TABLE 7.2. Size (in GB) and number of edges in provenance graphs

	Input	Local out	Global out
Size (GB)	28.43	8.11	1.47
Num of edges (millions)	127.73	32.98	17.13

shown in the table, backward provenance query takes 26.8 seconds on full provenance and only 2.3 seconds on reduced provenance. Then we selected a forward provenance query for an input tweet which consists of three hashtags. The forward query in our Twitter data application does not show a big difference in execution time as a given tweet only contains a limited number of hashtags and contributes only to that number of output data products. However, on a different application in which a single input contributes to a large number of outputs, reduced provenance would show much less forward query time compared to full provenance due to reduced number of edge traversals in the graph.

Table 7.2 shows the comparison of size (in GB) and the number of edges (in millions) among input, local output and global output provenance graphs generated for the full Twitter dataset of 10.1 GB with vertical partitioning and local batch size set to 20000. The results show that our parallel stream processing solution leads to a size reduction ratio of 19.34 and edge reduction ratio of 7.46 between the input and output provenance graphs while preserving backward and forward provenance.

7.2.2. Infinite provenance stream evaluation. We use a stream processing DIC workflow (see Figure 7.3) built for the same Twitter data processing use case to evaluate our provenance stream processing solution against an infinite provenance stream. As we discussed above, in this evaluation, the global reducer is configured to flush its reduced state periodically. When the ingress provenance stream is infinite, it's important to make sure

that the throughput of the provenance stream processing system is not below the ingress provenance throughput. In other words, if the stream consumers are slower than stream producers, consumers fall behind on the stream and it becomes harder to catch up. Therefore, in all our experiments below, we have scaled the provenance stream processing system to have the required level of throughput.

Environment. Both the raw Twitter stream and the provenance stream are ingested into the Kafka cluster. Eight nodes are allocated for the Kafka cluster where the master node runs the zookeeper instance and all eight nodes run Kafka brokers. Both Twitter stream processing job and the provenance processing job run on Flink. Total of 13 nodes is allocated for the Flink streaming cluster with one master node and all others acting as slaves. Each Flink node runs up to six stream consumers (one consumer per CPU core). Another node is allocated for Komadu and older provenance data are periodically cleaned to make sure it does not run out of disk space.

Workload. The workload a is streaming DIC which processes a raw stream of tweets, see Figure 7.2. The provenance stream is generated by the tweet processing stream consumers in parallel to the provenance stream as shown in Figure 7.4. As the throughput of the raw stream of tweets from the Twitter public API is too low for the purposes of our experiments, we use a Twitter dataset collected over time to replay as a stream at a higher ingress throughput. We developed a client which replays the twitter stream at a configured throughput.

Our experiments for infinite provenance streams are focused on fine-tuning the global reducer to maximize the efficiency of the system. In the first experiment, we measure the

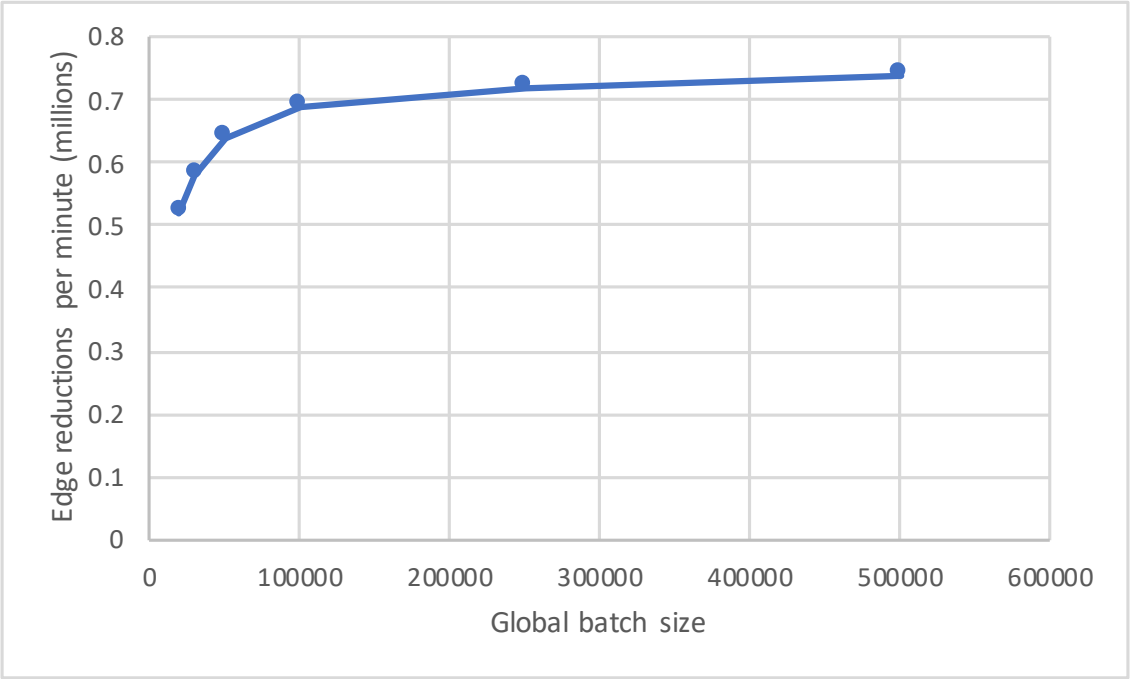


FIGURE 7.8. number of reductions (per minute) by the global reducer against the global batch size

number of reductions performed (within a unit of time) by the global reducer against the global batch size. Ingress throughput of the raw stream of tweets was set to 15 MB/s. Twitter stream processors generated a provenance stream with a throughput of around 34 MB/s. From our experiments above for finite provenance stream, we know that the local reducers can support up to 6 MB/s throughput with a local batch size of 20000. Therefore, we set the parallelism of the provenance streaming system to 8 so that it can support up to 48 MB/s throughput in total which is more than enough to process the ingress provenance throughput without falling behind on the stream. We used vertical partitioning for this experiment to make sure the throughput on each partition is roughly similar. As shown in Figure 7.8 number of reductions increase rapidly up to the batch size of 100000 and then does not show much improvement.

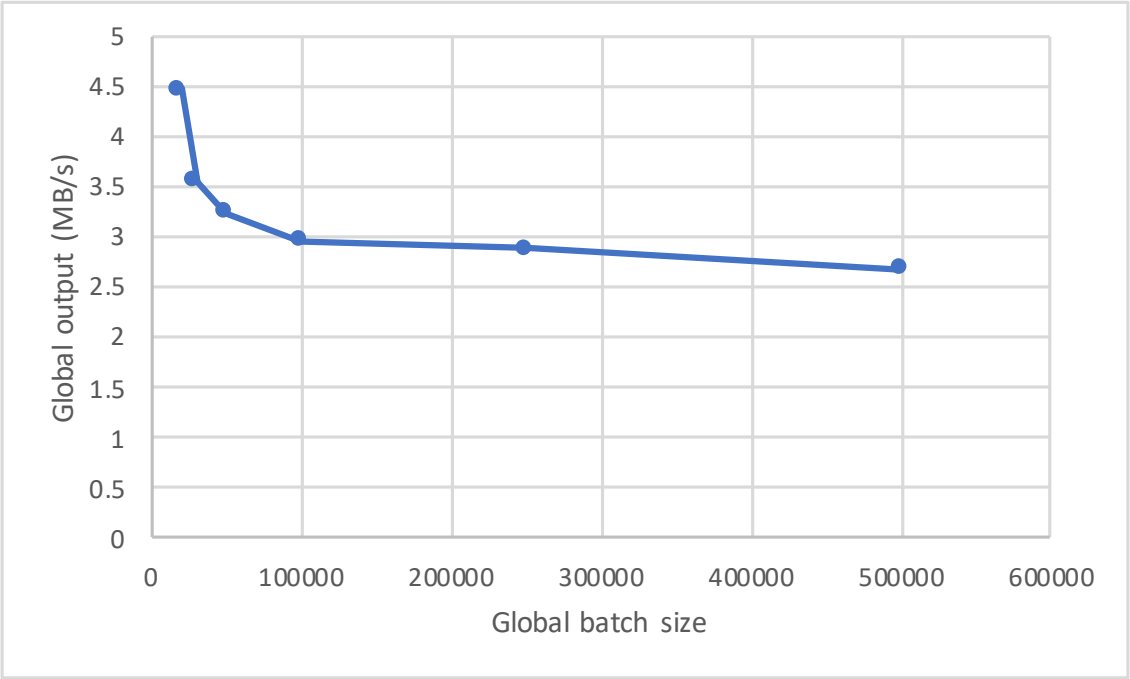


FIGURE 7.9. Output reduced provenance throughput (MB/s) emitted by the global reducer against the global batch size

In the second experiment, we measure the size of reduced provenance (within a unit of time) emitted by the global reducer against the global batch size. All configurations are kept same as the first experiment. Figure 7.9 shows the results and again shows a similar pattern to the first experiment above as the output size depends on the number of reductions.

CHAPTER 8

Conclusion and Future Work

In this thesis, we explore stream processing techniques to analyze a stream of provenance generated by a data-intensive computation. Our streaming solution reduces a provenance stream on-the-fly while preserving backward and forward provenance relationships. We present and evaluate a parallel, one-pass streaming algorithm designed for seamless horizontal scalability. This research results in an in-depth study which addresses the challenges in applying real-time stream processing techniques for a graph stream of provenance. Our work presents different approaches to partition a provenance stream and evaluates their performance under different types of streaming applications. Furthermore, it presents techniques to address out-of-order stream elements without compromising the accuracy of the results.

Our big provenance work draws on the Komadu provenance capture system and its application in a Data Lake prototype where data-intensive computations are used for continuous data processing. Volumes of fine-grained provenance data generated by data-intensive computations have proved to be multiple times larger than the input data, leading to issues in provenance data storage and query complexity. Our objective is to utilize stream processing techniques to process provenance streams in near real-time while moving analysis performed on stored provenance to on-the-fly techniques.

Laying a foundation for broader on-the-fly provenance analyses in the context of DICs, we focus on near real-time provenance reduction through preservation of backward and forward provenance. Our streaming architecture and the techniques are applicable in other types of provenance analyses as well. One is provenance stream partitioning which is critical for any type of provenance stream analysis. Stream element ordering issues are also common across most analyses and our research evaluates multiple techniques to address it. Furthermore, our solution for the *early elimination problem* is also important for algorithms which are based on summarizing a provenance graph.

We evaluated two techniques to solve the *early elimination problem*: Grouping and Sliding Window. Watermarks [**4**] [**3**] are an alternate method for solving the same problem and it is an important future work to evaluate it against the techniques we presented. Watermarks provide a mechanism to track the event time at the source of events within the stream processing system. The stream producer includes watermarks in the provenance stream from time to time to indicate the propagation of event time. Such watermarks can be used by the stream operators to decide whether there are more elements (related to a single function execution) to arrive.

There remain a number of interesting open questions in this work. The literature has only a few studies of applicability and utility of backward and forward provenance. Our system guarantees the preservation of this subset of provenance. It is interesting future work to formalize the breadth of use of this subset of provenance collected from DIC suites. Such an analysis will further help in optimizing such streaming solutions. There may be other types of provenance that should be combined with backward and forward provenance,

such as to improve reproducibility to address the question of *how necessary and sufficient is forward and backward provenance in supporting workflow reproducibility in DICs?*

Our streaming architecture uses two levels of stream processors: local reducers and a global reducer. What would be the benefit of extending this architecture to multiple-levels. When the number of stream partitions increases, a multi-level solution may lead to better throughput as the parallelism of the system can be further increased. However, the degree of reduction within initial levels can be less and that may contribute to higher propagation delays. *How can introduction of multiple-levels of architecture optimize the throughput?*

All our provenance stream partitioning strategies are based on the constraint that "all provenance edges generated during a single function execution must belong to the same partition in the stream". This constraint is introduced to ensure that valid dependencies are not lost during partitioning. *Lift constraint of membership while preserving all relationships within a function execution.*

There are worst case scenarios to be evaluated. As we use stateful stream processing, efficiency and degree of reduction of the system depend on the throughput of each local processor. When multiple local processors and a global processor are applied on an unbounded provenance stream, the degree of reduction within each batch of the global reducer can be maximized by having uniform throughput from local reducers. Imagine a situation where one local processor fails or slows down and lags behind other local processors to emit results. Now the reductions within the global processor can be impacted by this delay in receiving adjacent edges. *Extend system for fault tolerance.*

There are a number of *additional real-time provenance stream analyses use cases that can be carried out, for instance, detecting slow nodes in a compute cluster.* In a DIC, each function is applied on its input data in parallel using multiple compute nodes and the output is passed into the next function in the sequence. In most cases, next function execution cannot be started till all parallel tasks of the current function running on multiple nodes have been completed. Therefore, one slow node in the cluster can slow down the entire computation. Offline methods [**94**] to detect such slow nodes have been studied in the literature. But near real-time detection of slow nodes based on live provenance events can help to fix issues immediately.

Another use case for real-time provenance stream analysis is anomaly detection in computations. Few recent studies [**39**] [**105**] have highlighted some security vulnerabilities in the MapReduce programming paradigm mainly due to inadequate authentication and control over the worker nodes. Liao, C. et al. [**60**] use stored provenance information to build a solution for this problem. Our streaming architecture could be extended to use live provenance streams to detect anomalies in near real-time.

We use the combination of file path and byte offset as the unique identifier of a data product. When using backward or forward provenance, these identifiers are assumed to be unchanged so that the correct data products can be traced. However, this ID may not be globally unique in all cases of Data Lake implementation. This could be addressed by recording provenance on workflows (also known as prospective provenance) which moves data sets and integrating them with DIC provenance. *Extend for persistence and preservation of products of workflow.*

Some DIC systems, such as Spark, come with in-built provenance collection support. Our work currently collects provenance using its own techniques. *Integrate DIC default provenance into our provenance streaming model.* As our provenance streaming model does not depend on the origin or capture method of provenance, it should be possible to integrate self-generated provenance from DICs.

The products of this research are available at the following Git repositories:

- https://github.com/Data-to-Insight-Center/streaming-prov
- https://github.com/Data-to-Insight-Center/komadu

Bibliography

[1] J. Abraham, P. Brazier, A. Chebotko, J. Navarro, and A. Piazza. Distributed storage and querying techniques for a semantic web of scientific workflow provenance. In *2010 IEEE International Conference on Services Computing*, pages 178–185, July 2010.

[2] Charu C Aggarwal, Yuchen Zhao, and Philip S Yu. On clustering graph streams. In *Proceedings of the 2010 SIAM International Conference on Data Mining*, pages 478–489. SIAM, 2010.

[3] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: Fault-tolerant stream processing at internet scale. *Proc. VLDB Endow.*, 6(11):1033–1044, August 2013.

[4] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proc. VLDB Endow.*, 8(12):1792–1803, August 2015.

[5] Sherif Akoush, Ripduman Sohan, and Andy Hopper. Hadoopprov: Towards provenance as a first class citizen in mapreduce. In *Proceedings of the 5th USENIX Workshop on the Theory and Practice of Provenance*, pages 11:1–11:4. USENIX Association, 2013.

[6] Research Data Alliance. Data type registries working group. Available: `https://www.rd-alliance.org/groups/data-type-registries-wg.html`.

[7] Research Data Alliance. Pid information types working group. Available: `https://www.rd-alliance.org/groups/pid-information-types-wg.html`.

[8] H. Alrehamy and C. Walker. Personal data lake with data gravity pull. In *Big Data and Cloud Computing (BDCloud), 2015 IEEE Fifth International Conference on*, pages 160–167, Aug 2015.

[9] Ayman Alserafi, Alberto Abelló, Oscar Romero, and Toon Calders. Towards information profiling: Data lake content metadata management. In *Data Mining Workshops (ICDMW), 2016 IEEE 16th International Conference on*, pages 178–185. IEEE, 2016.

[10] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock. Kepler: an extensible system for design and execution of scientific workflows. In *Scientific and Statistical Database Management, 2004. Proceedings. 16th International Conference on*, pages 423–424, June 2004.

[11] Ilkay Altintas, Oscar Barney, and Efrat Jaeger-Frank. Provenance collection support in the kepler scientific workflow system. In *Provenance and annotation of data*, pages 118–132. Springer, 2006.

[12] Manish Kumar Anand, Shawn Bowers, and Bertram Ludäscher. Techniques for efficiently querying scientific workflow provenance graphs. In *EDBT*, volume 10, pages 287–298, 2010.

[13] ASF. Apache axis2. Available: `http://axis.apache.org/axis2/java/core`.

[14] ASF. Apache falcon. Available: `https://falcon.apache.org`.

[15] ASF. Apache flume. Available: `https://flume.apache.org`.

[16] ASF. Apache hadoop. Available: `http://hadoop.apache.org`.

[17] ASF. Apache hive. Available: `https://hive.apache.org`.

[18] ASF. Apache nifi. Available: `http://nifi.apache.org`.

[19] ASF. Apache oozie. Available: `http://oozie.apache.org`.

[20] ASF. Apache pig. Available: `https://pig.apache.org`.

[21] ASF. Apache samza. Available: `https://samza.apache.org`.

[22] Ziv Bar-Yossef, Ravi Kumar, and D Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 623–632. Society for Industrial and Applied Mathematics, 2002.

[23] Sean Bechhofer, David De Roure, Matthew Gamble, Carole Goble, and Iain Buchan. Research objects: Towards exchange and reuse of digital knowledge. 2010.

[24] Peter Buneman, Adriane Chapman, and James Cheney. Provenance management in curated databases. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 539–550, New York, NY, USA, 2006. ACM.

[25] Peter Buneman, Sanjeev Khanna, and Tan Wang-Chiew. *Why and Where: A Characterization of Data Provenance*, pages 316–330. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[26] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[27] Moses Charikar, Liadan O'Callaghan, and Rina Panigrahy. Better streaming algorithms for clustering problems. In *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, pages 30–39. ACM, 2003.

[28] Peng Chen, Tom Evans, and Beth Plale. *Analysis of Memory Constrained Live Provenance*, pages 42–54. Springer International Publishing, Cham, 2016.

[29] James Cheney, Amal Ahmed, and Umut a. Acar. Provenance as dependency analysis. *Mathematical. Structures in Comp. Sci.*, 21(6):1301–1337, December 2011.

[30] M. Chessell, F. Scheepers, N. Nguyen, R. van Kessel, and R. van der Starre. Governing and managing big data for analytics and decision makers, 2014.

[31] Fernando Chirigati, Dennis Shasha, and Juliana Freire. Reprozip: Using provenance to support computational reproducibility. In *Presented as part of the 5th USENIX Workshop on the Theory and Practice of Provenance*, 2013.

[32] Planet Lab Community. Planet lab. Available: `https://www.planet-lab.org`.

[33] Graham Cormode and S Muthukrishnan. Space efficient mining of multigraph streams. In *Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 271–282. ACM, 2005.

[34] Daniel Crawl, Jianwu Wang, and Ilkay Altintas. Provenance for mapreduce-based data-intensive workflows. In *Proceedings of the 6th Workshop on Workflows in Support of Large-scale Science*, pages 21–30. ACM, 2011.

[35] Susan B. Davidson and Juliana Freire. Provenance and scientific workflows: Challenges and opportunities. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 1345–1350. ACM, 2008.

[36] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[37] DOI.org. Didital object identifier system. Available: `http://www.doi.org`.

[38] Guozhu Dong, Leonid Libkin, Jianwen Su, and Limsoon Wong. Maintaining transitive closure of graphs in sql. *International Journal of Information Technology*, 51(1):46, 1999.

[39] J. Dyer and N. Zhang. Security issues relating to inadequate authentication in mapreduce applications. In *2013 International Conference on High Performance Computing Simulation (HPCS)*, pages 281–288, July 2013.

[40] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: A runtime for iterative mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 810–818, New York, NY, USA, 2010. ACM.

[41] Jeremy Fischer, Steven Tuecke, Ian Foster, and Craig A. Stewart. Jetstream: A distributed cloud infrastructure for underresourced higher education communities. In *Proceedings of the 1st Workshop on The Science of Cyberinfrastructure: Research, Experience, Applications and Models*, SCREAM '15, pages 53–61, New York, NY, USA, 2015. ACM.

[42] I. Foster, J. Vockler, M. Wilde, and Yong Zhao. Chimera: a virtual data system for representing, querying, and automating data derivation. In *Scientific and Statistical Database Management, 14th International Conference*, pages 37–46, 2002.

[43] Juliana Freire, Cláudio T Silva, Steven P Callahan, Emanuele Santos, Carlos E Scheidegger, and Huy T Vo. *Managing rapidly-evolving scientific workflows*, pages 10–18. Springer, 2006.

[44] Ashish Gehani, Minyoung Kim, and Tanu Malik. Efficient querying of distributed provenance stores. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 613–621, New York, NY, USA, 2010. ACM.

[45] Ashish Gehani and Dawood Tariq. Spade: support for provenance auditing in distributed environments. In *Proceedings of the 13th International Middleware Conference*, pages 101–120. Springer-Verlag New York, Inc., 2012.

[46] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SIGOPS operating systems review*, volume 37, pages 29–43. ACM, 2003.

[47] Boris Glavic. Big data provenance: Challenges and implications for benchmarking. In *Revised Selected Papers of the First Workshop on Specifying Big Data Benchmarks - Volume 8163*, pages 72–80. Springer-Verlag New York, Inc., 2014.

[48] Alon Halevy, Flip Korn, Natalya F Noy, Christopher Olston, Neoklis Polyzotis, Sudip Roy, and Steven Euijong Whang. Managing google's data lake: an overview of the goods system. *Data Engineering*, page 5, 2016.

[49] Handle.Net. Handle.net registry. Available: `http://handle.net`.

[50] Thomas Heinis and Gustavo Alonso. Efficient lineage tracking for scientific workflows. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1007–1018. ACM, 2008.

[51] Thomas Heinis and Gustavo Alonso. Efficient lineage tracking for scientific workflows. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1007–1018, New York, NY, USA, 2008. ACM.

[52] David A Holland, Uri Jacob Braun, Diana Maclean, Kiran-Kumar Muniswamy-Reddy, and Margo I Seltzer. Choosing a data model and query language for provenance. 2008.

[53] Robert Ikeda, Hyunjung Park, and Jennifer Widom. Provenance for generalized map and reduce workflows. 2011.

[54] Matteo Interlandi, Kshitij Shah, Sai Deep Tetali, Muhammad Gulzar, Seunghyun Yoo, Miryung Kim, Todd D. Millstein, and Tyson Condie. Titian: Data provenance support in spark. *PVLDB*, 9(3):216–227, 2015.

[55] Doug James, Nancy Wilkins-Diehr, Victoria Stodden, Dirk Colbry, Carlos Rosales, Mark Fahey, Justin Shi, Rafael F Silva, Kyo Lee, Ralph Roskies, et al. Standing together for reproducibility in large-scale computing: Report on reproducibility@ xsede. *arXiv preprint arXiv:1412.5557*, 2014.

[56] Knowledgent. How to design a successful data lake. Available: `http://knowledgent.com/whitepaper/design-successful-data-lake/`.

[57] I. Kouper, Y. Luo, I. Suriarachchi, and B. Plale. Provenance enriched pid kernel information as oai-ore map replacement for sead research objects. In *2017 ACM/IEEE Joint Conference on Digital Libraries (JCDL)*, pages 1–2, June 2017.

[58] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, pages 1–7, 2011.

[59] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 239–250, New York, NY, USA, 2015. ACM.

[60] C. Liao and A. Squicciarini. Towards provenance-based anomaly detection in mapreduce. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 647–656, May 2015.

[61] Tanu Malik, Ashish Gehani, Dawood Tariq, and Fareed Zaffar. *Sketching Distributed Data Provenance*, pages 85–107. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[62] Emaad A Manzoor, Sadegh Momeni, Venkat N Venkatakrishnan, and Leman Akoglu. Fast memory-efficient anomaly detection in streaming heterogeneous graphs. *arXiv preprint arXiv:1602.04844*, 2016.

[63] Ryan McConville, Weiru Liu, and Paul Miller. Vertex clustering of augmented graph streams. In *Proceedings of the 2015 SIAM International Conference on Data Mining*, pages 109–117. SIAM, 2015.

[64] Andrew McGregor. Graph stream algorithms: a survey. *ACM SIGMOD Record*, 43(1):9–20, 2014.

[65] Archan Misra, Marion Blount, Anastasios Kementsietsidis, Daby Sow, and Min Wang. Provenance and annotation of data and processes. chapter Advances and Challenges for Scalable Provenance in Stream Processing Systems, pages 253–265. Springer-Verlag, Berlin, Heidelberg, 2008.

[66] P. Missier, B. Ludascher, S. Bowers, S. Dey, A. Sarkar, B. Shrestha, I. Altintas, M.K. Anand, and C. Goble. Linking multiple workflow provenance traces for interoperable collaborative science. In *Workflows in Support of Large-Scale Science (WORKS), 2010 5th Workshop on*, pages 1–8, Nov 2010.

[67] Paolo Missier, Khalid Belhajjame, and James Cheney. The w3c prov family of specifications for modelling provenance metadata. In *Proceedings of the 16th International Conference on Extending Database Technology*, pages 773–776. ACM, 2013.

[68] L. Moreau, P. Missier, K. Belhajjame, R. B'Far, J. Cheney, S. Coppens, S. Cresswell, Y. Gil, P. Groth, G. Klyne, T. Lebo, J. McCusker, S. Miles, J. Myers, S. Sahoo, and C. Tilmes. Prov-dm: The prov data model. Available: `https://www.w3.org/TR/2013/REC-prov-dm-20130430/`.

[69] Luc Moreau, Ben Clifford, Juliana Freire, Joe Futrelle, Yolanda Gil, Paul Groth, Natalia Kwasnikowska, Simon Miles, Paolo Missier, Jim Myers, Beth Plale, Yogesh Simmhan, Eric Stephan, and Jan Van den Bussche. The open provenance model core specification (v1.1). *Future Gener. Comput. Syst.*, 27(6):743–756, June 2011.

[70] Luc Moreau and Paul Groth. Provenance: an introduction to prov. *Synthesis Lectures on the Semantic Web: Theory and Technology*, 3(4):1–129, 2013.

[71] Luc Moreau, Bertram Ludäscher, Ilkay Altintas, Roger S Barga, Shawn Bowers, Steven Callahan, George Chin, Ben Clifford, Shirley Cohen, Sarah Cohen-Boulakia, et al. Special issue: The first provenance challenge. *Concurrency and computation: practice and experience*, 20(5):409–418, 2008.

[72] Kiran-Kumar Muniswamy-Reddy, David A Holland, Uri Braun, and Margo I Seltzer. Provenance-aware storage systems. In *USENIX Annual Technical Conference, General Track*, pages 43–56, 2006.

[73] J. Myers, M. Hedstrom, D. Akmon, S. Payette, B. A. Plale, I. Kouper, S. McCaulay, R. McDonald, I. Suriarachchi, A. Varadharaju, P. Kumar, M. Elag, J. Lee, R. Kooper, and L. Marini. Towards sustainable

curation and preservation: The sead project's data services approach. In *2015 IEEE 11th International Conference on e-Science*, pages 485–494, Aug 2015.

[74] J Myers, Carmen Pancerella, Carina Lansing, K Schuchardt, Brett Didier, Naveen Ashish, and Carole A Goble. Multi-scale science, supporting emerging practice with semantically derived provenance. In *ISWC workshop on Semantic Web Technologies for Searching and Retrieving Scientific Data*. Florida, 2003.

[75] Tom Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Martin Senger, Mark Greenwood, Tim Carver, Kevin Glover, Matthew R. Pocock, Anil Wipat, and Peter Li. Taverna: A tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, pages 3045–3054, 2004.

[76] Wellington Oliveira, Daniel de Oliveira, and Vanessa Braganholo. Experiencing prov-wf for provenance interoperability in swfmss. In *Provenance and Annotation of Data and Processes: 5th International Provenance and Annotation Workshop, IPAW 2014*, pages 294–296. Springer, 2014.

[77] Hyunjung Park, Robert Ikeda, and Jennifer Widom. Ramp: A system for capturing and tracing provenance in mapreduce workflows. In *37th International Conference on Very Large Data Bases (VLDB)*. Stanford InfoLab, August 2011.

[78] Christoph Quix, Rihan Hai, and Ivan Vatov. Metadata extraction and management in data lakeswith gemms. *Complex Systems Informatics and Modeling Quarterly*, (9):67–83, 2016.

[79] W. Sansrimahachai, L. Moreau, and M. J. Weal. An on-the-fly provenance tracking mechanism for stream processing systems. In *2013 IEEE/ACIS 12th International Conference on Computer and Information Science (ICIS)*, pages 475–481, June 2013.

[80] W. Sansrimahachai, M. J. Weal, and L. Moreau. Stream ancestor function: A mechanism for fine-grained provenance in stream processing systems. In *2012 Sixth International Conference on Research Challenges in Information Science (RCIS)*, pages 1–12, May 2012.

[81] Atish Das Sarma, Sreenivas Gollapudi, and Rina Panigrahy. Estimating pagerank on graph streams. *Journal of the ACM (JACM)*, 58(3):13, 2011.

[82] Carlos Scheidegger, David Koop, Emanuele Santos, Huy Vo, Steven Callahan, Juliana Freire, and Cláudio Silva. Tackling the provenance challenge one layer at a time. *Concurrency and Computation: Practice and Experience*, 20(5):473–483, 2008.

[83] Amazon Web Services. Kinesis data streams. Available: `https://aws.amazon.com/kinesis`.

[84] Paul Shannon, Andrew Markiel, Owen Ozier, Nitin S Baliga, Jonathan T Wang, Daniel Ramage, Nada Amin, Benno Schwikowski, and Trey Ideker. Cytoscape: a software environment for integrated models of biomolecular interaction networks. *Genome research*, 13(11):2498–2504, 2003.

[85] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*, pages 1–10. IEEE, 2010.

[86] Yogesh Simmhan, Paul Groth, and Luc Moreau. Special section: The third provenance challenge on using the open provenance model for interoperability. *Future Generation Computer Systems*, 27(6):737–742, 2011.

[87] Yogesh L. Simmhan, Beth Plale, and Dennis Gannon. A survey of data provenance in e-science. *SIGMOD Rec.*, 34(3):31–36, September 2005.

[88] Yogesh L Simmhan, Beth Plale, and Dennis Gannon. Query capabilities of the karma provenance framework. *Concurrency and Computation: Practice and Experience*, 20(5):441–451, 2008.

[89] Manolis Stamatogiannakis, Hasanat Kazmi, Hashim Sharif, Remco Vermeulen, Ashish Gehani, Herbert Bos, and Paul Groth. Trade-offs in automatic provenance capture. In *Proceedings of the 6th International Provenance and Annotation Workshop on Provenance and Annotation of Data and Processes - Volume 9672*, IPAW 2016, pages 29–41, Berlin, Heidelberg, 2016. Springer-Verlag.

[90] Brian Stein and Alan Morrison. The enterprise data lake: Better integration and deeper analytics. Available: `https://www.pwc.com/us/en/technology-forecast/2014/cloud-computing/assets/pdf/pwc-technology-forecast-data-lakes.pdf`.

[91] Isuru Suriarachchi and Beth Plale. Crossing analytics systems: A case for integrated provenance in data lakes. In *e-Science (e-Science), 2016 IEEE 12th International Conference on*, pages 349–354. IEEE, 2016.

[92] Isuru Suriarachchi and Beth Plale. Provenance as essential infrastructure for data lakes. In *International Provenance and Annotation Workshop*, pages 178–182. Springer, 2016.

[93] Isuru Suriarachchi, Quan Zhou, and Beth Plale. Komadu: A capture and visualization system for scientific data provenance. *Journal of Open Research Software*, 3(1), 2015.

[94] Jiaqi Tan, Xinghao Pan, Eugene Marinelli, Soila Kavulya, Rajeev Gandhi, and Priya Narasimhan. Kahuna: Problem diagnosis for mapreduce-based cloud computing environments. In *Network Operations and Management Symposium (NOMS), 2010 IEEE*, pages 112–119. IEEE, 2010.

[95] Y. Tas, M. J. Baeth, and M. S. Aktas. An approach to standalone provenance systems for big social provenance data. In *2016 12th International Conference on Semantics, Knowledge and Grids (SKG)*, pages 9–16, Aug 2016.

[96] Ignacio Terrizzano, Peter M Schwarz, Mary Roth, and John E Colino. Data wrangling: The challenging yourney from the wild to the lake. In *CIDR*, 2015.

[97] Ignacio Terrizzano, Peter M Schwarz, Mary Roth, and John E Colino. Data wrangling: The challenging yourney from the wild to the lake. In *CIDR*, 2015.

[98] Name to Thing. Archival resource key. Available: `http://n2t.net/e/ark_ids.html`.

[99] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. Storm@twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 147–156. ACM, 2014.

[100] Alvaro Videla and Jason JW Williams. *RabbitMQ in action*. Manning, 2012.

[101] Nithya Vijayakumar and Beth Plale. Tracking stream provenance in complex event processing systems for workflow-driven computing. In *EDA-PS Workshop*, 2007.

[102] Nithya N. Vijayakumar and Beth Plale. Towards low overhead provenance tracking in near real-time stream filtering. In *Proceedings of the 2006 International Conference on Provenance and Annotation of Data*, IPAW'06, pages 46–54, Berlin, Heidelberg, 2006. Springer-Verlag.

[103] Jianwu Wang, D. Crawl, S. Purawat, Mai Nguyen, and I. Altintas. Big data provenance: Challenges, state of the art and opportunities. In *Big Data (Big Data), 2015 IEEE International Conference on*, pages 2509–2516, Oct 2015.

[104] Jianwu Wang, Daniel Crawl, and Ilkay Altintas. Kepler+hadoop: A general architecture facilitating data-intensive applications in scientific workflow systems. In *Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science*, pages 12:1–12:8. ACM, 2009.

[105] W. Wei, J. Du, T. Yu, and X. Gu. Securemr: A service integrity assurance framework for mapreduce. In *2009 Annual Computer Security Applications Conference*, pages 73–82, Dec 2009.

[106] Tobias Weigel, Timothy DiLauro, and Thomas Zastrow. Pid information types wg final deliverable. 2015.

[107] A. Woodruff and M. Stonebraker. Supporting fine-grained data lineage in a database visualization environment. In *Proceedings 13th International Conference on Data Engineering*, pages 91–102, Apr 1997.

[108] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, pages 10–10. USENIX Association, 2010.

[109] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438, New York, NY, USA, 2013. ACM.

[110] Dongfang Zhao, Chen Shou, Tanu Malik, and Ioan Raicu. Distributed data provenance for large-scale data-intensive computing. In *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*, pages 1–8. IEEE, 2013.

[111] Jun Zhao, Carole Goble, Robert Stevens, and Sean Bechhofer. Semantically linking and browsing provenance logs for e-science. In *Semantics of a Networked World. Semantics for Grid Databases*, pages 158–176. Springer, 2004.

ISURU SURIARACHCHI

isurues@gmail.com

PROFESSIONAL EXPERIENCE

Software Engineer **Amazon, Seattle** **Jan 2018 - Present**

- A key member of the AWS Kinesis Data Streams front end team.
- Made several contributions in adding HTTP/2 streaming support for Kinesis consumers which reduced the end-to-end propagation delay of a data record by 60%.
- Currently working on a project to improve the scalability of the Kinesis front end fleet.

Summer Intern **Amazon, Seattle** **Summer 2016**

- Worked with the AWS Kinesis Data Streams team.
- Implemented a new streaming channel for Kinesis based on Reactive Sockets specification for producing and consuming streams from customer applications.

Summer Intern **Amazon, Seattle** **Summer 2015**

- Worked with the AWS EC2 Networking team.
- Designed and developed a distributed service which maintains the mappings between physical machines in the EC2 network and the component which controls them.

Senior Technical Lead **WSO2, Sri Lanka** **May 2008 – July 2012**

- Worked as a core developer of the WSO2 Carbon framework, the OSGi based middleware platform for the entire WSO2 product stack.
- Lead developer of the WSO2 Application Server (May 2010 to July 2012), a Java runtime for hosting Web Services and Web Applications.
- Implemented lazy loading for tenants and services in WSO2 Stratos (Platform as a Service on Cloud).
- Implemented WS-Policy aware Caching and Throttling components for WSO2 Application Server.
- Integrated Apache CXF into WSO2 Application Server to support JAX-WS and JAX-RS service deployment.
- Worked onsite for many customers in USA and Europe and architected and developed SOA solutions using WSO2 middleware product stack.

Research Assistant **Indiana University, Bloomington** **Aug 2012 – Dec 2017**

- Designer and lead developer of the Komadu provenance repository.
- Implemented a Map-Reduce based solution on Azure Cloud for executing large number of storm surge simulations as a contribution to the SLOSH project which was a collaboration research work with the National Hurricane Center.
- Made major contributions for SEAD project which is a research data preservation framework.

Open Source Contributor **Apache Software Foundation** **Sep 2006 – Present**

- Committer and PMC member of Apache Web Services project.
- Implemented JSON message format support for Apache Axis2, which is a Java Web Services engine.
- Implemented Hierarchical Service deployment support for Apache Axis2.
- Contributed to Apache Chainsaw project, which is a remote log monitoring tool under Log4j.

EDUCATION

PhD, Computer Science **Nov 2018**

Indiana University, Bloomington, IN

Dissertation Title: Big Provenance Stream Processing for Data Intensive Computations

GPA: 3.982

MSc, Computer Science **Jan 2016**
Indiana University, Bloomington, IN

BSc, Computer Science and Engineering **April 2008**
University of Moratuwa, Colombo, Sri Lanka

SKILLS

- Distributed Systems, Distributed Stream Processing, Cloud Computing, Application Servers, SOA and Web Services, Data Provenance, Provenance Repositories
- Java, C, Python, Scheme, PHP, JavaScript, JSP, SQL, XML, WSDL, WS-BPEL, HTML
- MySQL, MongoDB, Cassandra, OSGi, Git, Subversion, Maven, Ant, JSON, SOAP, REST, Apache Hadoop, Apache Spark, Apache Tomcat, Apache Axis2, RabbitMQ, Linux, Windows

ACADEMIC PROJECTS

- **Advanced OS** (Fall 2013): Implemented a Shell on top of Linux system call APIs. Implemented the main OS components on top of the embedded Xinu operating system on Raspberry Pi.
- **Computer Networks** (Fall 2012): Implemented a port scanner in C comparable to Nmap. Supported scanning techniques like TCP SYN scan, TCP ACK scan etc.
- **Scientific Data Management** (Spring 2013): Did a performance comparison of Cassandra and MongoDB by ingesting a large amount of data and generated analytics using Map-Reduce.
- **Knowledge Based AI** (Spring 2013): Built a SLOSH instance partitioning tool to optimize the total running time on cloud using Case Based Reasoning.
- **Algorithm Design and Analysis** (Spring 2014): Implemented an algorithm in C to multiply large integers which cannot fit to a word in a computer. For example, multiplying two 1000-bit integers.
- **Undergraduate final year research** (May 2007 – April 2008): Rampart2, a high performance WS-Security module for Apache Axis2 which uses a pull parsing object model for XML. Improved invalid message rejection time by 70% using an efficient WS-SecurityPolicy validation algorithm.

PUBLICATIONS

- **Papers**
  - Isuru Suriarachchi, and Beth Plale. "Big Provenance Stream Processing for Data Intensive Computations." *e-Science (e-Science), 2018 IEEE 14th International Conference on*. IEEE, 2018.
  - Isuru Suriarachchi, and Beth Plale. "Crossing analytics systems: A case for integrated provenance in Data Lakes." *e-Science (e-Science), 2016 IEEE 12th International Conference on*. IEEE, 2016.
  - Isuru Suriarachchi, Quan Zhou, and Beth Plale. "Komadu: A capture and visualization system for scientific data provenance." *Journal of Open Research Software* 3.1 (2015).
  - Isuru Suriarachchi, and Beth Plale. "Provenance as essential infrastructure for Data Lakes." *International Provenance and Annotation Workshop*. Springer, Cham, 2016.
  - Abhirup Chakraborty, Milinda Pathirage, Isuru Suriarachchi, Kavitha Chandrasekar, Craig Mattocks, and Beth Plale. "Executing Storm Surge Ensembles on PAAS Cloud." *Cloud Computing for Data-Intensive Applications*. Springer, New York, NY, 2014.
- **Posters**
  - Suriarachchi, I., Survey of Provenance Practices in Data Preservation Repositories, RDA 5th Plenary Meeting, Amsterdam, Netherlands, Sep 2014.
  - Plale, B., Mattocks, C., Chakraborty, A., Chandrasekar, K., Pathirage, M., and Suriarachchi, I., SI2-SSE Pipeline Framework for Ensemble Runs on the Cloud, SI2 PI Meeting, DC, Jan 2013.

HONORS AND AWARDS

- Google Summer of Code **2007**
- Award for best undergraduate final year project, University of Moratuwa **2008**
- Gold medal for best academic performance, University of Moratuwa **2008**
- Merit award at the National Best Quality Software Awards for Rampart2 **2008**